1998

Improvements to the color quantization process

Rance David Necaise
College of William & Mary - Arts & Sciences

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

IMPROVEMENTS TO THE COLOR QUANTIZATION PROCESS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Rance David Necaise

1998

APPROVAL SHEET

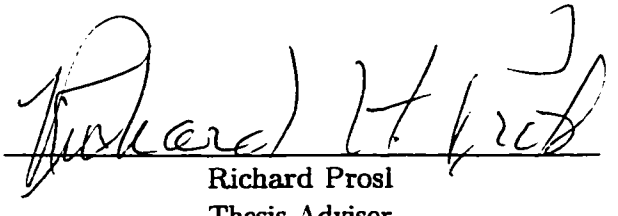
This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

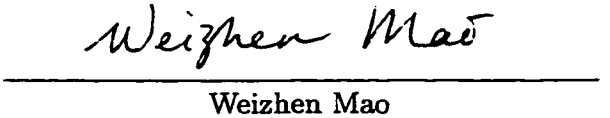
Rance D. Necaise

Approved, August 1998



Richard Prosl
Thesis Advisor

Stephen Park



Weizhen Mao

William Bynum

Sidney Lawrence
Department of Mathematics

To my parents ...

Contents

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Richard Prosl, for his guidance and patience throughout the creation of this dissertation. He was also a source of inspiration in my goal to become a good and effective teacher. I will be forever indebted to him. I would also like to thank the members of my committee for their time and assistance. I am also greatful to the entire faculty of the Computer Science department at the College of William & Mary and to my friends and colleagues at the University of Southern Mississippi.

It was a great pleasure to have the opportunity to teach at the College of William & Mary both as an instructor and teaching assistant. Thus, I would like to thank my many students that made it enjoyable and who unknowingly provided the encouragement I needed to complete this endeavor.

I have had the opportunity to make many friends at William & Mary. They have provided support, encouragement, and fun times. While they are too numerous to list, I must acknowledge some of them: Barry Lawson for his humor and his willingness to always help; Dr. Jesson Havill, the person to whom I could complain about anything and he understood completely; Felipe Perrone for the encouragement and countless discussions about nothing in particular; Phil Auld and Joel Hollingsworth for their system expertise; Mark Idema, a great volleyball coach; Dr. Matt Johnson for being; Karen Anewalt, my faithful lunch buddy; Dawn Idema, my original cycling buddy; Kia and Gregg Rippel, for her source of endless conversation and his attitude on the volleyball court; and the countless others including Sharon and Dennis Edwards, Dr. Anna Brunstrom, Beth Havill, and Heidi Hollingsworth.

Finally, I wish to thank my family for their love and support. I especially wish to thank my parents, Willard and Ella Rea Necaise, who have always supported me in whatever I chose to pursue.

List of Figures

ABSTRACT

The presentation of color images on devices with limited color capabilities requires a reduction in the number of colors contained in the images. Color image quantization is the process of reducing the number of colors used in an image while maintaining its appearance as much as possible. This reduction is performed using a color image quantization algorithm. The quantization algorithm attempts to select k colors that best represent the contents of the image. The original image is then recolored using the representative colors. To improve the resulting image, a dithering process can be used in place of the recoloring.

This dissertation deals with several areas of the color image quantization process. The main objective, however, is new or improved algorithms for the production of images with a better visual quality than those produced by existing algorithms while maintaining approximately the same running time. First, a new algorithm is developed for the selection of the representative color set. The results produced by the new algorithm are better both visually and quantitatively when compared to existing algorithms. Second, a new nearest-neighbor search algorithm that is based on the Locally Sorted Search algorithm is developed to reduce the time required to map the input colors to a representative color. Finally, two modifications are made to the error-diffusion dithering technique that improve the execution time. These modifications include the use of a two-weight matrix for the distribution of the error values and the presentation of a method to parallelize the error-diffusion technique. Furthermore, the analytical results of several experiments are provided to show the effectiveness of each of these additions and improvements.

IMPROVEMENTS TO THE COLOR QUANTIZATION PROCESS

Chapter 1

Introduction

The processing of a large data set frequently requires that its elements be paired with those of a smaller, usually finite, carefully chosen set of representatives. An example is the pairing of a set of real numbers with a set of integer values. This reduction is known by different names depending upon the area in which it is performed. In general, it is known as *quantization*.

Many authors originally used the term quantization to refer to the conversion of signals from analog to digital [27, 66, 107]. Today, the term is often used to refer to any reduction of a large data set to a smaller, usually finite one. Another term that is analogous to quantization and is more common in some areas is data compression. The term data compression, however, generally refers to the reduction of any type of data set which can usually be uncompressed to its original state. Quantization, generally refers to the reduction of specific sets of data in such a way that the original data can not be obtained by a reverse operation.

1.1 Background

The most common uses of quantization involve the storage, transmission, and processing of large data sets. The transmission of data includes the broadcasting of signals over the public air waves and connection oriented networks. Bandwidth reduction of television broadcast signals falls in this category. Researchers continue to study methods for reducing the amount of information that must be transmitted to produce high quality images on television sets. Another example is the quantization of signals transmitted from satellites that have been launched to the far reaches of our solar system [78]. Today, quantization is also used to reduce or compress digital data to speed its transmission. Communications between computers can be improved by compressing the data before it is transmitted over the telephone network.

There are many uses of quantization in image processing [15]. Two of these areas include edge detection and image classification. Images that are to be evaluated by computers or human observers, such as those from satellites, can be improved by detecting and highlighting edges of objects contained in the image. This detection helps to improve the perception of detail since "the edges generally indicate the physical extent of objects" [32]. Quantization can improve some edge detection algorithms by reducing the number of intensities, especially around the edges [15].

In image classification, objects or patterns in an image can be detected and classified. This classification can be aided by first reducing the number of colors in the image. With fewer colors to work with, less sophisticated classification algorithms are needed; there are fewer decisions for the computer to make [15, 78].

Computer graphics and multimedia applications use quantization techniques with color images in which results are needed in real time [15]. Voice recognition, finger print matching, and computer vision applications also rely on quantization techniques [15, 32].

1.2 Quantization Process

The process of pairing elements of a large set with those of a smaller set of representatives is called *quantization*. The quantization process is accomplished by a quantizer, a definition for which ends the following string of definitions.

Definition 1.1 *Let S be some scalar or vector space and let $P = \{p_1, p_2, \ldots, p_k\}$ be a finite set of subsets of S. P is a* **k-partition** *of the set S iff: (1) $p_i \subseteq S$ and $p_i \neq \emptyset$, for all $1 \leq i \leq k$, (2) $p_i \cap p_j = \emptyset$, for all $1 \leq i < j \leq k$, and (3) $\bigcup_{1 \leq i \leq k} p_i = S$. P is a* **partial partition** *if only conditions (1) and (2) are satisfied.*

Definition 1.2 *Suppose P is a partition of S. The elements of P are called the* **clusters** *of the partition.*

Definition 1.3 *Let $P = \{p_1, p_2, \ldots, p_k\}$ be a partition of S and let $Y = \{y_1, y_2, \ldots, y_k\}$ be a subset of S such that $y_i \in p_i$, for all $1 \leq i \leq k$. $Q_{P,Y}$ is a* **quantizer** *of S, iff: $Q_{P,Y}$ is a function on S onto Y such that for all $1 \leq i \leq k$, $x \in p_i \Rightarrow Q_{P,Y}(x) = y_i$. We write Q instead of $Q_{P,Y}$ since the relevant partition and* **representative values** *(***code book***), Y, are usually understood in context.*

A quantizer designed to work with scalar data is known as a *scalar quantizer*. Those designed for use with multidimensional or vector data, are known as *vector quantizers* [52].

1.3 Optimal Solution

The result of any quantizer Q is the replacement of an input value x from a large set by a representative y from a smaller set. For a given input set of values, R, there will be an inherent distortion or difference between the original and replacement values. This difference is called the *quantization error*. The quantization error can be described in either local or global terms. Local quantization error is a measurement of the difference between a single input value and the representative value to which it maps. Global quantization error is a measurement over the entire input set R. The quantization error is measured using analytical methods which are known as *quantization error metrics* [52, 54, 59].

In general, a quantization algorithm attempts to design a quantizer such that the global quantization error is minimized. A quantizer that will produce the absolute minimum global quantization error is known as the *optimal quantizer*; the result of such a quantizer is known as the *optimal solution* [52]. It is known, however, that the problem of finding the optimal solution is NP-complete [11, 24, 28, 52, 108]. Thus, solutions are rarely provably optimal [82, 110].

The many quantization algorithms in use can be divided into two groups: iterative and heuristic. The *iterative* methods attempt to minimize the local quantization error by iteratively improving the set of representative values and partition until some threshold is met. This process can be time consuming; the iterative methods tend to compromise speed for quality [82, 110].

The heuristic methods, on the other hand, compromise quality for speed. Heuristic algorithms tend to find an "acceptable" solution very quickly [28, 82, 110]. In general,

heuristic methods iteratively subdivide the domain until k clusters have been formed. These k clusters become the basis for the selection of the representative set, Y, and partition, P, that form the quantizer.

1.4 Iterative Quantization Methods

The most well known iterative quantization scheme is that due to Lloyd [54] and Max [59]. The goal of the Lloyd-Max problem is to minimize the mean-square difference between the set of input scalar values and the resulting set of output values.

The Lloyd-Max solution constructs a quantizer by starting with a finite set of input values $A = \{a_1, a_2, \ldots, a_n\}$ and an initial set of representative values $Y = \{y_1, y_2, \ldots, y_k\}$ such that $k \ll n$. The partition, $P = \{p_1, p_2, \ldots, p_k\}$, is constructed by mapping each input value x to the representative value, y_i, which produces the smallest mean-square difference $(x - y_i)^2$ for all $1 \leq i \leq k$. The input value becomes a member of the cluster corresponding to the representative to which it mapped. Thus, if input value x maps to the representative y_i, then x becomes a member of the i^{th} cluster.

After partitioning the space, a new representative value set is constructed by computing the average value for each cluster

$$y_i = \frac{1}{|p_i|} \sum_{x \in p_i} x, \quad 1 \leq i \leq k.$$

This process is then iterated for a given number of times or until there is little or no change in the average mean-square differences computed for consecutive representative sets.

The selection of the initial set of representatives, from which a partition can be constructed, is commonly done in one of two ways. Either a set of representative values can

be selected at random or by some statistical means such as the k most frequently occurring values [55, 82]. The initial set of representatives is only meant to be an approximation used to start the iterative process.

In 1980, Linde et. al. [52] extended this algorithm for use with vector data. This extension is often referred to as the LBG algorithm (named for its authors) [82]. Other variations of the Lloyd-Max problem have been proposed [31, 67, 106, 107].

1.5 Color Image Quantization

Color image quantization is the process of reducing the number of colors needed for the display or storage of an image, while maintaining, as much as possible, the appearance of the original. Algorithms designed for use with color images are a form of vector quantization and are known as color image quantization algorithms or simply *color quantization algorithms.*

Color image quantization is a form of image compression. In order to save storage space, the digital image is compressed by reducing the number of colors used in that image. This type of image compression is a lossy algorithm. That is, the compressed image can not be reprocessed to generate the original image.

Color image quantization is needed to display digital images on some devices. Color monitors and color printers are very popular today. Most of these devices are limited in the number of colors that can be shown at one time. Many images that need to be displayed are composed of far more colors than these devices can handle, Therefore, it is imperative to have a method for reducing the number of colors contained in an image in order to display that image on a device with limited color capabilities.

The image quantization problem is not only concerned with the construction of a quantizer but also with its use. After a quantizer is designed, it is normally used in one of two ways. The most common is to simply replace the input image by mapping each input color to a representative color. I term this technique *recoloring*. The second method uses a spatial integration technique to create the illusion of varying shades of colors where none actually exists. This spatial integration uses the same quantizer that would be used with the recoloring technique. The only difference is that the colors or a small area of colors in the original image are manipulated before they are recolored. This spatial integration technique is commonly termed *dithering*[1].

1.6 Motivation

There are several applications in graphics and multimedia where speed is very important in the manipulation of images. The most common ones today include the world wide web and multimedia applications. With the explosion of the world wide web and multimedia applications, the transmission and viewing of color images has become commonplace. However, many computers and workstations still use limited display devices. Thus, some form of quantization must be performed in order to view these images.

These applications rely on the presentation of information in real time. Much of that information includes color images. Thus, there is little time to spend on the quantization and dithering of a single image.

Given the need for speed, iterative color quantization algorithms are seldom used in interactive types of applications. The decision is commonly made in favor of an algorithm

[1]Dithering or spatial integration techniques are described in detail in Chapter 5.

that can produce good results very quickly rather than better results using a time consuming iterative algorithm.

The driving force behind my research has been the need for quicker algorithms that could be used with interactive type of applications. This includes both the design and use of the quantizer. Thus, I focused on the development and use of heuristic quantization algorithms. The main focus was on the production of images with a better visual quality than those produced by existing algorithms while maintaining approximately the same running time.

The work for this dissertation began with a study of several heuristic color quantization algorithms. After making minor improvements to several algorithms, I turned my attention to the inclusion of the properties of human vision in the quantization and dithering processes. It soon became obvious, however, that the inclusion of the more significant properties would result in time consuming algorithms. Thus, I turned my attention to reducing the actual running times of the more popular color quantization algorithms. During my work, I developed a new quantization algorithm and a new solution to the nearest-neighbor search algorithm. I also improved the Floyd-Steinberg dithering algorithm for color images to reduce the execution time.

1.7 Thesis Overview

This dissertation is divided into six chapters. The first chapter introduced the quantization process and provided a brief overview. The second chapter provides an overview of quantization error metrics that are used to measure the analytical quality of quantization algorithms. During my research, I discovered that the most popular metric should not be

used to measure or compare the quality of quantizing an image when dithering. Thus, I researched and found a metric that can be used with both recoloring and dithering.

In Chapter 3, I introduce the popular heuristic image quantization algorithms. This introduction is by way of a review which includes both fixed and adaptive algorithms. I then introduce a new quantization algorithm (the color cut algorithm) that I developed. I show, using analytical results, that my algorithm produces better results than those produced by the other algorithms. In addition, I provide a modification that can be made to my algorithm for use on computers with limited memory.

A new solution to the nearest-neighbor problem is introduced in Chapter 4. Most heuristic color quantization algorithms, do not actually partition the color space for use with the quantizer. Instead, they simply map each unique input color to a representative color when the quantizer is used. The most common mapping scheme is to map an input color to its nearest-neighbor. This introduces a special case of the classic nearest-neighbor problem for which Heckbert [35] introduced a solution. During my research, I discovered that Heckbert's idea could be expanded to improve the search time. In Chapter 4, I provide an overview of Heckbert's algorithm and then introduce my new algorithm.

Chapter 5 introduces the dithering and halftoning techniques used for both gray-level and color images. Dithering can greatly improve the visual results when quantizing a color image. In this chapter, I show that the popular Floyd-Steinberg dithering technique can be modified for color images to improve the running time. I also show that the results of this modification do not degrade the images to which they are applied.

Finally, in Chapter 6 I provide a conclusion of my contributions introduced in the three previous chapters. In addition, I provide a list of several items that warrant future study.

Chapter 2

Quantitative Image Difference

2.1 Introduction

The result of color quantization is an image which contains fewer colors than the original. It is desirable for the resulting quantized image to resemble the original image as closely as possible.

How can we rank order a collection of color images which have been quantized from a single original? Surely one method is to do a subjective ranking in which human obse-vers are asked to rank order the images based solely upon visual perception [49, 58, 62, 78, 103]. Such orderings by human observers will be "nearly" the same if the processed images differ greatly. But this tends not to be true when the processed images are similar [58, 78].

The results of a subjective test can vary depending upon the test conditions and the type of observers used. The type of hardware used for the experiments and the physical lighting conditions of the room can have a dramatic affect on the results [33, 78]. In addition, if trained "expert" observers are used, the results may differ from those produced by "non

expert" observers. It is assumed that observers experienced in processing or analyzing images can detect small degradations that may be overlooked by "non experts" [78].

Finally, some sets of images, such as those represented in the RGB color space using 10-bits per color component and those in the infra-red part of the spectrum, can not be displayed at all so human observers simply can not be used. But it is necessary to classify the quality of quantizing such images.

What we need are *quantization error metrics* – analytical methods for measuring differences between two images. The main objective of any analytical metric is to measure how well a quantized image resembles the original. It is desirable that the metric produce results that match, to a significant extent, those of a subjective test [58, 62, 78].

In this chapter, I review definitions of quantization error metrics for both gray-level and color images. With this background, I explore one problem with the root-mean-square error metric and introduce a second metric that solves that problem.

2.2 Previous Work

The original work on quantization error metrics focused on continuous and discrete gray-level images. Given the motivation for this thesis, however, we are only concerned with those for discrete images.

2.2.1 Metrics for Gray-Level Images

The general approach in measuring the quantization error is to measure the "difference" between corresponding pixel values of the original and quantized images [15, 78, 105]. The most common error measures are the *mean-square* and *root-mean-square* metrics [15, 54,

59, 78, 105]. Given two scalar images, I and Q, each of size $m \times n$, the mean-square error is given by[1]

$$\mathrm{MS}' = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} (I[i,j] - Q[i,j])^2$$

and the root-mean-square error by

$$\mathrm{RMS}' = \sqrt{\mathrm{MS}'}.$$

Some authors use versions of these metrics that compute relative errors instead of the absolute error [71, 78]. Two of the more common include computing the root-mean-square relative to the average pixel value or the standard deviation of the original image. The average pixel is computed by

$$\hat{\alpha}(I) = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} I[i,j]$$

while the standard deviation of an image is given by

$$\hat{\sigma}^2(I) = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} (I[i,j] - \hat{\alpha}(I))^2.$$

Other common metrics include the L_1-*norm* and the L_∞-*norm* [31, 71]

$$\mathrm{L}_1(I - Q) = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} |I[i,j] - Q[i,j]|$$

[1]Some of the following equations have versions for gray-level and color images. These equations are given the same name and can be identified by the context in which they are used. In addition, some equations have an absolute and relative version. These two versions are distinguished by using an ′ to indicate the absolute version.

$$L_\infty(I-Q) = \max_{[m,n]} |I[m,n] - Q[m,n]| .$$

The *covariance* and *correlation* of an image can also be used as a quantization error metric [71, 78]. The covariance between two discrete gray-level images is given by

$$\gamma(I,Q) = \frac{1}{mn}\sum_{i=1}^{n}\sum_{j=1}^{m}(I[i,j] - \hat{\alpha}(I))(Q[i,j] - \hat{\alpha}(Q))$$

while the correlation is given by

$$\rho(I,Q) = \frac{\gamma(I,Q)}{\hat{\sigma}(I)\hat{\sigma}(Q)}.$$

The *Minkowski distance* (or Holder norm) [16], a generalization of the root-mean-square metric, has become more popular in recent years. This metric is given by

$$L_E(I-Q) = \left(\frac{1}{mn}\sum_{i=1}^{n}\sum_{j=1}^{m}|I[i,j] - Q[i,j]|^E\right)^{\frac{1}{E}}$$

for some value of E.

There is also a group of metrics that transform or manipulate the gray-level values of the pixels in each image before the error is computed. The purpose of transforming the pixel values is an attempt to compensate for some of the properties of human vision so the results will more closely resemble those of a subjective test. The most common of these is the *normalized point-transformed mean-square* error [78],

$$\text{TNMS} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}[p(I,i,j) - p(Q,i,j)]^2}{\sum_{i=1}^{n}\sum_{j=1}^{m}p(I,i,j)^2}$$

with either a power law [104] or logarithmic function [58, 104] used for the pixel transformation, i.e.,

$$p(I,x,y) = I[x,y]^v \qquad \text{or} \qquad p(I,x,y) = c_1 \log_b (c_2 + c_3 I[x,y]) .$$

Finally, a discrete convolution kernel can be used in the computation of the mean-square or root-mean-square error and in turn introduce another error metric [71, 78]. The discrete convolution operation constructs a new image by computing a new value for each pixel. These new values are the weighted averages between a pixel value and its neighbors. In the simplest form, the computations are a weighted average of the pixel values in a small area with the original pixel at the center of this area [88].

The weights used to compute the new pixel values are given as an $l \times l$ matrix, commonly termed a *convolution kernel.* The sample 3×3 kernel illustrated in Figure 2.1 would be a factor in the computation of an average value for a given pixel over the 3×3 area surrounding that pixel.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.1: A sample 3×3 convolution kernel.

Given an $m \times n$ image, I, and an $l \times l$ convolution kernel, K, the computation of the new value for the pixel (x,y) is given by

$$g(I,x,y) = \frac{1}{w} \sum_{i=y-\delta}^{y+\delta} \sum_{j=x-\delta}^{x+\delta} (I[i,j] \cdot K[y-i, x-j]) \tag{2.1}$$

where w is the sum of the elements of K and δ is half the width of K and is defined as $\delta = \lfloor l/2 \rfloor$.

To use convolution in the context of an error metric, a weight matrix must be supplied. Given a weight matrix, the convolution operation is applied to both the original and quantized images. The *convolution mean-square* or *convolution root-mean-square* error is then computed for the convolved images [71]. The traditional way to perform this operation is to use function $g(\cdot)$ from equation (2.1) within the mean-square equation

$$\mathrm{CMS}'_a = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} (g(I,i,j) - g(Q,i,j))^2 .$$

2.2.2 Metrics for Color Images

Metrics have also been developed for use with color images. Many of these common metrics are simple extensions of those used with gray-level images, the only difference being that the computations are performed on image values representing three dimensional points in some color space.

The most common color error metric is the absolute mean-square error [33, 35, 71] using the RGB color space and is given by[2]

$$\mathrm{MS}' = \sum_{i=1}^{n} \sum_{j=1}^{m} d(I[i,j], Q[i,j]) \tag{2.2}$$

where the Euclidean distance is

$$d(\mathbf{x},\mathbf{y}) = (\mathbf{x}_0 - \mathbf{y}_0)^2 + (\mathbf{x}_1 - \mathbf{y}_1)^2 + (\mathbf{x}_2 - \mathbf{y}_2)^2 \tag{2.3}$$

[2]Symbols presented in boldface type indicate a 3×1 vector with access to an individual element indicated by a numeric subscript.

and $\mathbf{x}$ and $\mathbf{y}$ are points in a 3-D color space. The Manhattan distance [31]

$$d'(\mathbf{x},\mathbf{y}) = |\mathbf{x}_0 - \mathbf{y}_0| + |\mathbf{x}_1 - \mathbf{y}_1| + |\mathbf{x}_2 - \mathbf{y}_2|$$

and the RMS error [71]

$$\mathrm{RMS}' = \sqrt{\mathrm{MS}'}$$

can also be used with color images.

While the absolute mean-square error is the most common in the literature, a relative version might be preferred. As with the gray-level metrics, the standard deviation of the original image can be used to compute the relative error

$$\mathrm{MS} = \frac{\mathrm{MS}'}{\sigma(I)} \qquad \text{and} \qquad \mathrm{RMS} = \frac{\mathrm{RMS}'}{\sigma(I)}$$

where

$$\sigma^2(I) = \sum_{i=1}^{n}\sum_{j=1}^{m} d(I[i,j], \alpha(I)) \tag{2.4}$$

and $\alpha(I)$ is the average color of the source image I and is a point in the color space.

Though these metrics are the most popular, they do not take into account the properties of human vision, especially when using the RGB color space. Thus, some authors have recommended the use of more perceptual color spaces such as CIE-Lab, CIE-Luv or YIQ [89]. These color spaces have been designed to correspond more analytically to the human perception of color than the RGB color space [63, 78, 104].

The development of a metric for use with color images is more difficult than that for gray-level images because of the many factors of human vision that play a role. In recent years, some authors have studied metrics which directly take into account some of the properties of human vision including contrast sensitivity and spatial masking [50, 58, 65], the spectral response of the human eye [50, 83], and Weber's law which concerns the sensitivity of the human eye to a change in luminance that is proportional to the background's luminance [38].

2.3 Problems with the Root-Mean-Square Error

In general, some image can be found to contradict the credibility of any error metric [78]. Thus, special metrics must be developed to work with special problems.

During my work for this thesis, I worked with many different images and quantized versions of those images[3]. I used the root-mean-square error (RMS) relative to the standard deviation to measure the difference between the original and quantized images. I discovered, however, that the visual results consistently contradicted the results of the RMS error when using dithered images. This contradiction occurred with all 50 images.

The most likely reason the RMS square error did not work in these comparisons is that it only considers the difference between corresponding pixel values. It does not take into account the spatial integration performed by the human eye. It is well accepted that the human eye can not distinguish the color of small individual dots without being influenced by the color(s) of the surrounding area [20, 72, 81]. Thus, the human eye tends to blend the colors over an area to create the illusion of different shades of colors. This idea of spatial

[3]The quantized versions were created using heuristic color quantization algorithms with recoloring and dithering. Heuristic algorithms are discussed in Chapter 3; dithering in Chapter 5.

integration allows for the reproduction of images with different shades of gray on a bi-level device using a technique known as dithering [20, 33]. Dithering algorithms rely on this blending of colors to create more visually pleasing images.

Consider the group of images[4] in Figure 2.2. The top image is the original. The image in the middle has been quantized and recolored to reduce the number of colors from 6642 to 64. The bottom image has been dithered using the Floyd-Steinberg dithering technique to reduce the contouring affect. The same colormap was used for this process as with the recoloring process.

From a visual inspection (from a printed version and on a display device) the dithered image more closely resembles the original than the recolored. However, the RMS error difference (0.054 for the recolored image and 0.066 for the dithered) indicates just the opposite. Similar results occurred for all 50 images in the set of test images. Thus, the RMS error metric is not a viable metric when comparing images that have been dithered.

2.4 The Convolution Root-Mean-Square Error

To compensate for the lack of spatial integration by the RMS error metric, I propose the use of the convolution root-mean-square error metric with color images using an approximation of the Gaussian distribution as the kernel. I will refer to this metric as the $CRMS_g$ error metric.

[4]Some artifacts in these images are introduced by the technique used by laser printers. However, the striking differences between these images correspond to that of a visual inspection on a display device.

(a)

(b)

(c)

Figure 2.2: Sample color images (a) original with 6642 colors, (b) recolored version with 64 colors, and (c) dithered version with 64 colors.

The use of the convolution operation takes into account the spatial integration performed by the eye. The result is a blending or smoothing affect over the entire image before the error is computed. This reduces the sharpness between adjacent pixels and tends to make the change in intensity more gradual from one pixel to the next [88].

In image processing, the Gaussian distribution is approximated using the the following kernel, with $w = 17$,

$$K = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

2.4.1 The Construction of the Metric

Given a color image, the convolution kernel must be applied in turn to each component of the color of a given pixel. If a 3-D color point is represented as a 3×1 vector, then $\mathbf{h}(x, y)$ computes the new color for a given pixel where $g(\cdot)$ is the equation in (2.1) and the subscript indicates the color component

$$\mathbf{h}(I, x, y) = \begin{bmatrix} g_0(I, x, y) \\ g_1(I, x, y) \\ g_2(I, x, y) \end{bmatrix}.$$

Given an original image, I, and a quantized image, Q, the equation in (2.2) is modified to work with a color image and function $\mathbf{h}(\cdot)$ from above

$$\text{CMS} = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} d(\mathbf{h}(I, i, j), \mathbf{h}(Q, i, j))$$

where function $d(\cdot)$ is the distance equation from (2.3). Given the convolution mean-square error for two color images, the CRMS_g is then computed relative to the standard deviation

of the original image

$$\mathrm{CRMS}_g = \frac{\sqrt{\mathrm{CMS}}}{\sigma(I)}. \tag{2.5}$$

The standard deviation and average color are also computed using the weighted average colors. Thus, equation (2.4) must be modified

$$\sigma^2(I) = \sum_{i=1}^{n}\sum_{j=1}^{m} d(\mathbf{h}(I,i,j), \alpha(I)).$$

The edges of the images present a special problem for the CRMS_g metric. In this case, blind application of the formula for the computation of the average color would require the use of pixel values which lie outside the boundaries of the image. One solution to this problem is to write code in such a way that no attempt is ever made to access pixel values where pixels do not exist. Another solution is to write code which uses a fixed pixel (black, white, the average color of the image) whenever attempts are made to access values of non-existing pixels. I use the latter solution with a pixel value of $(0,0,0)$. My experiments have shown that my results are not influenced by the choice of pixel values as long as the same choice is used for the original image as for the quantized one.

2.4.2 Motivation

The motivation for the use of the convolution root-mean-square error metric with dithered images is derived from the work of MacIntyre and Cowan [57]. As stated above, it is well accepted that the human eye blends or integrates the colors over an area to create the illusion of different shades of colors. Thus, when we view an image, we do not see the colors

of individual pixels. Instead, we see a blend of colors in which the color of each pixel is influenced by its neighbors. This is commonly referred to as *pixel bleed.*

MacIntyre and Cowan studied the pixel bleed effect (or influence of nearby pixels) on the computation of the luminance contrast on a display device. In their work, they were able to calculate the influence of neighboring pixels and the area of this influence. They showed that the contribution of neighboring pixels in the computation of the luminance contrast could be expressed as a Gaussian function centered at the reference pixel. From this, the percentage contributed by each neighbor to the reference pixel was computed. These percentages are shown in the diagram in Figure 2.3.

•	•	•	•	•
•	2.5%	10.7%	2.5%	•
•	10.7%	**47.0%**	10.7%	•
•	2.5%	10.7%	2.5%	•
•	•	•	•	•

Figure 2.3: A matrix illustrating the influence of neighboring pixels on the center pixel in the computation of the luminance over the area of 25 pixels. The pixels marked with a • contribute little to this computation.

The reference pixel is in the center cell of the diagram. It only contributes 47 percent to the luminance in this area of 25 pixels. The immediate horizontal and vertical neighbors contribute more than the immediate diagonal neighbors. The pixels on the outer edge of the area (represented by a •) contribute less than 0.1 percent.

Pappas and Neuhoff [70] use this approach to improve halftone images for presentation on a laser printer. They use this model to compute how dark a given spot in the halftone image will appear using a particular dot pattern.

Chapter 3

Heuristic Color Quantization Algorithms

3.1 Introduction

Heuristic color quantization algorithms attempt to approximate a solution to the general quantization problem. They were originally designed for use with color digital images and have become known as color image quantization algorithms. While some color quantization algorithms have greatly compromised quality for speed, others have attempted to balance speed and quality.

In this chapter, I present a formal definition of the heuristic color quantization problem and some of the details in the design of a heuristic color quantizer. This is followed by a review of the commonly used heuristic algorithms; specifically those for use in interactive applications.

Finally, I present a new color quantization algorithm which produces better results than the common algorithms, both visually and analytically. The analytical quality of my algorithm is supported using the results of several experiments conducted to compare it with existing algorithms.

3.1.1 Problem Formulation

The color quantization problem is derived from the general vector quantization problem. It is commonly solved using either an iterative or heuristic type algorithm. If the solution is needed in real-time, a heuristic algorithm is generally chosen. There are two major differences between the heuristic and iterative algorithms; these differences account for the popularity of the heuristic algorithms in real-time applications. First, the heuristic methods include the selection of an initial partition as part of their solution. Second, the heuristic methods do not attempt to improve the design by repeatedly partitioning the space and selecting a new representative set. The lack of the iterative step contributes to the improved speed of the heuristic methods.

The heuristic quantization algorithms can be divided into two categories: fixed and adaptive. The fixed algorithms design a quantizer independent of the image to be quantized. Once designed, they can be applied to any image. The adaptive algorithms, on the other hand, rely upon the contents of the image. Thus, a new quantizer must be constructed for each image. As with the general quantization problem, the major part of the quantization process is the design of the quantizer. In the following sections I review some of the more popular heuristic quantization algorithms. First, some definitions and notation used with color digital images.

Definition 3.1 *Let X be the set of integers within the range $[0 \ldots 2^8)$. Then* **RGB** $= \{(r, g, b) \mid r \in X, g \in X, b \in X\}$ *is the standard discrete 24-bit three dimensional RGB color space. An element of* RGB *is called a* **color** *and is denoted as (r, g, b), where $r, g,$ and b are color component values.*

Definition 3.2 *Suppose that m and n are integers. An* **image** *is an $m \times n$ matrix of colors.*

We commonly refer to an element of the image as a **pixel**. In addition, we commonly refer to the color stored as an element of the image as the color of the pixel when it really is the color of the pixel that would be displayed in a rendering of the image.

Definition 3.3 *Given $a, b \in$ RGB, $dist(a, b)$ is the Euclidean distance between a and b.*

Definition 3.4 *The representative set of* RGB *values used by a quantizer is also known as a* **colormap** *or a* **representative color set**.

3.1.2 Heuristic Quantizer Design

The design process of the heuristic algorithms consists of the three steps illustrated in Figure 3.1. First, the RGB color space[1] is subdivided to create an initial partition, P'. The result of this partitioning step may actually be a partial partition created from a subset of the RGB color space. Next, a representative color set, Y, is constructed from the initial partition. Finally, a new partition, P, is created from the representative set constructed in the previous step.

[1] Authors normally suggest the use of the RGB color space since it is the most common for the construction and storage of images. Some work has been done in the area of image quantization using alternate color spaces [25, 100]. When used with images stored as RGB values, however, a conversion to the alternate color space must be performed. This conversion routinely requires the use of floating-point arithmetic which is not suitable for real-time quantization applications.

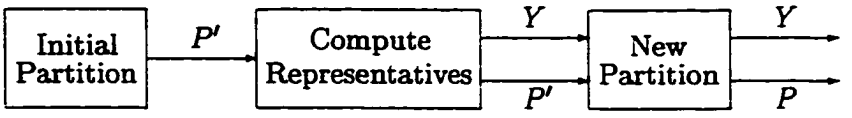


Figure 3.1: The Heuristic quantizer design process.

Most of the algorithms only differ in the method used to construct the initial partition. The other two steps are very much the same for most of the algorithms. To construct the representative color set, the average color or centroid of each cluster is computed. These colors then become the entries in the representative color set. The average color for a given cluster is computed component-by-component.

In creating the new partition, P, one of two methods is generally employed. Some algorithms simply reuse the initial partition (this reduces the construction time). Thus, there is a one-to-one correspondence between the clusters of P' and the elements of Y. That is, an element of cluster p'_i will be mapped to the representative color y_i where y_i is the average color computed for cluster p'_i. This method is known as *centroid mapping*. The second approach is to map each element of the initial partition, P', to a representative color based upon the minimum Euclidean distance. This approach, known as *nearest representative mapping*, is commonly used when the initial partition, P', is a partial partition of the RGB color space. If x is an element of the subset of RGB which P' partitions, x becomes a member of p_j, where $p_j \in P$, if and only if $dist(x, y_j) \leq dist(x, y_i)$ for all $1 \leq i \leq k$, and $i \neq j$.

3.2 Previous Work

3.2.1 Fixed Quantization Algorithms

The simplest quantization algorithms are those which design the quantizer independent of the image to be processed [25, 27, 29, 100]. The general approach is to create the initial partition by subdividing the RGB color cube into equal slices along each dimension to produce approximately k cubes. The result is a partition of the RGB color space in which each cluster is a cube. The creation of the representative set is handled by computing the centroid of each cluster. The initial partition is then used by the quantizer as the final partition [33].

The major advantages of a fixed quantization algorithm is its speed and simplicity of implementation. This is evident from the fact that fixed heuristic quantization algorithms have a worst-case time complexity of $O(mn)$. These algorithms are useful when the goal of quantization is simply the use of color for image enhancement – pseudo color or false color image enhancement [33, 71].

The speed and simplicity of the fixed quantization algorithms, however, do not out-weigh the many drawbacks. The most important of which is the poor visual quality of the resulting image. Poor visual quality results in large part from the fact that many of the k colors go unused. Thus, when $k = 256$, if equal slices are used, the cube would be divided into six parts along each of the three dimensions with a result of only 216 subcubes. In the simplest algorithms, the remaining 40 entries are not used. To alleviate this problem, a second approach is to subdivide the color cube in varying number of slices along each dimension. A common approach (for $k = 256$) is to sub divide the red and green dimensions into eight

equal slices while subdividing the blue dimension into four equal slices [29, 73].

Another reason for the poor visual quality is that the colors in the original image may map to a small fraction of the k representative colors. This is especially true when the colors used in the image produce clusters in a small area of the RGB cube.

3.2.2 Adaptive Quantization Algorithms

The adaptive heuristic quantization algorithms create the initial partition of the RGB color space based on the distribution of colors in the original image. The result is a partial partitioning in which the elements of P' contain only those colors occurring in the image. Hence, these algorithms depend upon the contents of the original image for the design of the quantizer.

In most cases, an image contains many instances of the same colors [82]. In fact, if c is the number of unique colors in a given image, it is not uncommon for $c \ll mn$. To improve the processing time, most of the adaptive algorithms use a histogram (defined below) to represent the distribution of unique colors present in an image. The histogram is then used as the input to the quantizer design process instead of the actual image.

Definition 3.5 *Let I be an image as defined in Definition 3.2. A color* **histogram** *of I is a three dimensional matrix in which the entry at (r, g, b) is the count of the number of times the color (r, g, b) occurs in I.*

By creating the initial partition based on the distribution of colors in the image, we take into account the natural color distribution in most images and reduce the waste of representative colors that is common in the fixed algorithms. Those areas of the RGB color

cube in which colors of the image actually occur gain the focus for construction of the partition.

There are numerous adaptive heuristic quantization algorithms. In this section, I review several of the more commonly used and studied algorithms. It should be noted that future references to quantization algorithms in this thesis will refer to adaptive heuristic color quantization algorithms unless explicitly stated otherwise.

3.2.2.1 Popularity

The popularity algorithm [35] is the most basic of the adaptive quantization algorithms. The algorithm starts by scanning the image to produce a 3-D histogram of the image's color distribution. The k most frequently appearing colors are selected to be the entries in the representative color set Y. With the color set created, the final or new partition is created using the nearest representative mapping scheme.

The popularity algorithm performs poorly on images with a large number of colors simply because it fails to consider colors in the sparse regions of the color space. It is a good algorithm, however, for producing draft images in which exact detail is not the goal of the resulting image.

3.2.2.2 Median Cut

The *median cut* quantization algorithm, developed by Heckbert [35], performs much better than the popularity algorithm because it is "more sensitive to the color distribution of the original image." The basic idea is to let each entry in the representative color set Y represent approximately the same number of elements in the original image I.

The quantizer design phase of the median cut algorithm starts with a color histogram constructed from I. To reduce the storage requirements and the computation time, the algorithm reduces the number of possible unique colors in the histogram. This reduction, known as *bit reduction*, is performed by converting the three low order bits of each component to zero. In C++ syntax, the reduction can be defined as

```
struct rgb { byte r, g, b; };

rgb MedianCutReduction( rgb x )
{
   x.r = (x.r >> 3) << 3;
   x.g = (x.g >> 3) << 3;
   x.b = (x.b >> 3) << 3;
   return( x );
}
```

As the first step in creating the initial partition, the median cut algorithm constructs a list L from the non zero entries of the histogram. Each element of the list contains a color and the histogram count for that color.

The minimum-bounding box of L is computed and its longest dimension, ϵ, is determined. The minimum-bounding box is defined as follows

Definition 3.6 *Let* $L = \{l_1, l_2, \ldots, l_w\}$ *be a list where* $l_i \in$ RGB *for all* $1 \leq i \leq w$. *The* **minimum-bounding box** *of* L *is the smallest box which encloses all elements in* L*: it is identified by its lower-left and upper-right vertices.*

The colors in L are then sorted from smallest to largest using the ϵ-component as the sort key. The ordered list is then split at the item containing the median image element to create two sub lists. This entry is found by traversing the list from the front and counting the number of image elements containing each color. The entry at which approximately half

of all the image elements contain colors occurring before that entry's color in the ordered list is the entry containing the median image element. The original list is discarded. This process is repeated using the sublists until k such lists have been formed. For each iteration, one of the previously constructed sub lists is selected to be the next list split. The list chosen is the one containing the most pixels.

After the k lists have been created, each list becomes a cluster in the initial partition, P'. The representative set is created using the common construction scheme. To create the new partition, P, the median cut algorithm maps each color in P' (the reduced colors) to the nearest representative color in Y. Thus, P is a partial partition of the RGB space.

3.2.2.3 Center Cut

The *center cut* quantization algorithm [41] is based on Heckbert's median cut algorithm. It implements three modifications to the median cut's quantizer design process. The first modification is a change in the bit reduction scheme. The scheme chosen by Joy and Xiang is one which corresponds to the computation of the brightness component in the YIQ color space [20, 71], where emphasis is given to the green component. In C++ syntax, the center cut scheme is

```
rgb CenterCutReduction( rgb x )
{
   x.r = (x.r >> 3) << 3;
   x.g = (x.g >> 2) << 2;
   x.b = (x.b >> 4) << 4;
   return( x );
}
```

The second modification changes the position at which a sub list is split. Instead of splitting the list at the entry containing the median image element, the center cut algorithm

selects the mid point, μ, of the longest dimension, ϵ, and creates two sub lists L_1 and L_2 as follows: $L_1 = \{x \mid x \in L \wedge x_\epsilon \leq \mu\}$ and $L_2 = L - L_1$.

Finally, the last modification made by the center cut algorithm is in the selection of the next sub list to be split. Instead of selecting the list with the most pixels, the list with the "longest-longest dimension" is chosen. This is simply the longest dimension of all three dimensions of all the sub lists.

3.2.2.4 Octree Quantization

An *MX-octree* is described by Samet [86] as a variant of an *octree* [18, 20, 85, 86]. First, the *MX-octree* subdivides the three dimensional space into cubes representing single discrete points. The 3-D space represented by the octree has a lower bound of $(0, 0, 0)$ and an upper bound of $(2^d - 1, 2^d - 1, 2^d - 1)$, where d is some integer greater than zero. Each integer valued point within the bounding box of this space is represented by a leaf node in the octree. All leaf nodes are at the lowest depth and all data is stored in the leaf nodes. In an MX-octree, the data usually corresponds to information represented by the given point (i.e., a frequency count, descriptive information, etc.). The interior nodes indicate the splitting of the cube into octants. To conserve space, however, the nodes are created only when they are needed.

The octree quantization algorithm developed by Gervautz and Purgathofer [29] uses the octree data structure to create the initial partition of the quantizer. The actual octree structure used is a variant, part MX-Octree, part general octree.

The MX-octree used in the octree quantization algorithm represents the RGB color cube and has a depth $d = 8$. The data nodes contain two fields: a color vector and the color

frequency count. The color vector field may contain the color of an individual element of the image or it may contain the sum of the colors of several such elements.

Unlike many of the other heuristic algorithms, the octree quantization algorithm does not produce a histogram of the colors occurring in the image. Instead, it creates clusters of colors using the MX-octree structure. To create the initial partition, P', the image is traversed one row at a time. During this traversal, the color of each pixel is inserted into the tree. If a color was previously inserted, the count field of the corresponding leaf node is incremented and the color is added to the color vector field. When the $k+1$ leaf node is created, a merging step (described below) takes place to reduce the number of leaf nodes in the tree to be less than or equal to k. This merging step is performed each time the number of leaf nodes reaches $k+1$. Thus, at any given time, the octree will contain no more than k leaf nodes.

To perform the merging step, upon insertion of the $k+1$ color, an interior node at the lowest level is replaced by the average color of its leaf nodes. This interior node then becomes a leaf node. Once an interior node becomes a leaf node, it never returns to being an interior node. Thus, all interior nodes at the lowest level must be reduced before an interior node at the next highest level. This leads to the possibility of selecting an interior node for merging which has a single leaf node. The merging process is repeated until the number of leaf nodes is less than or equal to k.

After processing the entire image, the resulting tree is used as the initial partition since the path of every color in the image will lead to a leaf node. The representative color set is then constructed by averaging the color vectors in each leaf node. To create the new

partition, P, the initial partition is used (centroid mapping) producing a partial partition of the RGB color space.

3.2.2.5 Other Algorithms

The three quantization algorithms presented in the previous sections are commonly used for comparisons with new quantization algorithms. There are, however, numerous other color quantization algorithms in the literature.

The mean-split [102] and variance-based [102, 101] algorithms are also variants of the median cut algorithm. The mean-split algorithm attempts to split a list of colors at the average component of the longest bounding-box dimension. The variance-based method finds the point at which to split the list based on the minimization of the color variance. Variance values are computed for different positions along all three axes with the smallest of these chosen as the point and axis about which the list is split.

To improve upon the center cut and median cut algorithms, Roytman and Gotsman [82] developed the oct-cut quantization algorithm. Their algorithm uses an octree to store the colors from the image, after being reduced using a bit-cutting scheme similar to that of the median cut algorithm. The colors in the tree represent the initial cluster for which a bounding box is computed. The box is then split along each axis simultaneously at the center to form eight new boxes. Some of these boxes will be empty and thus are discarded. This process is repeated until k boxes have been formed. The oct-cut algorithm selects the next box to be split by choosing the box representing the largest number of pixels.

Xiang and Joy have a second algorithm which attempts to improve upon the results of the octree quantization algorithm. In the agglomerative clustering technique [110], they

attempt to construct clusters of colors by merging smaller clusters into larger ones until k clusters have been formed. Unlike many of the other techniques which start with a single cluster and split it into smaller clusters, this algorithm starts with many smaller clusters and attempts to merge them into the desired number of clusters.

The pairwise-nearest neighbor (PNN) algorithm [17] is another example of an algorithm which starts with a number of smaller clusters (the original colors) and attempts to merge them to form k clusters. Originally, no efficient method was provided to actually merge clusters to create larger clusters. In 1991, however, Balasubramanian and Allebach [4] devised an efficient method for the implementation of a modified version of the PNN algorithm.

The binary splitting algorithm [5] is one of the few algorithms which attempts to split larger clusters into smaller ones in a non-orthogonal manner. This algorithm uses the binary space partition tree [22, 94] to split color clusters along a plane chosen based upon the variance of the given cluster instead of perpendicular to one of the three axes.

3.3 The Color Cut Quantization Algorithm

The color cut algorithm starts by building a histogram of the colors in the input image. The actual colors in the image are stored in the histogram; no bit cutting is performed. A list is then constructed of those colors in the histogram for which the number of occurrences is greater than zero.

Given the list of colors, a list splitting operation is performed until k lists are created. This process begins by computing the minimum bounding box of the colors in the list. The colors in the list are then sorted using the longest dimension as the primary sort key. To

break a tie when two primary sort keys have the same value, the second longest dimension is used as the secondary key followed by the last dimension as the final sort key. If two of the dimensions have the same length, precedence is given to the green dimension, followed by the red, and concluding with the blue.

After sorting the colors, the list is split at the middle entry to create two sub lists. The two sub lists will contain approximately the same number of colors. The process is then repeated on the sub lists, in a breadth first fashion, until k color lists have been created.

After creating the k sub lists, each list becomes a cluster in the initial partial partition, P'. The average weighted color of each list becomes an entry in the representative set Y. The new partition, P, is then constructed using the nearest representative mapping scheme. The creation of P' and Y is illustrated by the following code

```
colormap ColorCutAlg( image I, int k )
{
   ColorList ListA, ListB;
   Histogram H;
   int nboxes, key[3];
   queue Q = ø;

   ListA = BuildHistogram( I, H );
   Enqueue( Q, ListA );
   nboxes = 1;
   while( nboxes < k ) {
      ListA = Dequeue( Q );
      ComputeSortKey( ListA, key );
      SortAndSplitList( ListA, ListB, key );
      Enqueue( Q, ListA );
      Enqueue( Q, ListB );
   }
   return( CreateColorMap( Q ) );
}
```

3.3.1 Building The Histogram

To build the histogram, we need a fast and memory efficient structure. The easiest approach would be to use a three dimensional array of size $256 \times 256 \times 256$, where each element of the array represents a single color. This approach, would require 256^3 array elements, most of which would go unused. On the other hand, the advantage of using an array is that accessing and updating the count of a particular color is direct.

A second approach is to use a binary search tree [7] in which each node represents a single color. For each pixel in the image, the tree is searched for the color of that pixel. If the color is already in the tree, a count field is incremented, otherwise, a new node is created for the color. The drawback to this approach is the search time required to find a color in the tree.

The approach I use is a combination of the two data structures described above (this approach is similar to one used by Balasubramanian et. al. [5]). I use a two dimensional array of size 256×256 where the rows correspond to the red color component and the columns correspond to the green color component. Each element of the array is a pointer to the root of a binary search tree (or an empty pointer when the tree is empty). The nodes within a binary search tree contain those colors having the same red and green components. Only the blue components differ and thus, are used to sort the nodes within the tree. The histogram structure is illustrated in Figure 3.2.

After the histogram is create, a list of the colors in the image is needed. To construct this list, a single linked list of search tree nodes is created. This list is constructed as the nodes are inserted into binary trees of the histogram. The binary tree nodes each contain

five data fields: the `color`, the count of the number of occurrences of the given color, links to the left and right child nodes and a link to the next node in the linked list of nodes. After constructing the histogram, the linked list becomes the first cluster. This cluster is then split to create two smaller clusters, and so on.

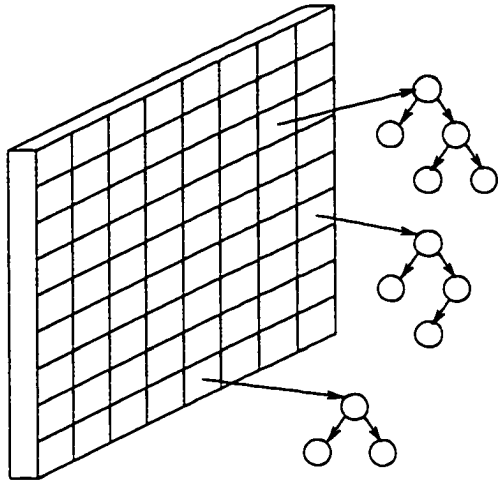


Figure 3.2: The histogram data structure using a 2-D array of binary search trees. Each binary tree contains all the colors having the same red and green components. The nodes in a given binary tree are ordered on the blue component.

3.3.2 Sorting and Splitting the List

The colors in the list are sorted in a sub list using three sort keys based on the dimensions of the minimum bounding box of that list. A fast method is needed to sort the list of colors. A suitable solution is to create a 1-D array large enough to hold all of the colors in the list and then sort the array using an $O(n \log n)$ sorting algorithm where n is the number of colors in the list.

However, many of the colors will have the same primary key color component. Thus, a lot of time will be wasted comparing these values. Likewise, the same argument could be

made, but to a lesser extent, about the secondary key. Thus, I propose the use of a special structure to combine the sorting and splitting of the list.

First, the colors in the list are divided into smaller lists based on the primary key. All colors having the same primary key value are placed in the same list. This sub division is handled quickly using an array of pointers to singly linked lists which I term the *sort array*.

The ordering of the colors in the linked lists is not important at this time. Thus, each color is inserted at the front of the list. The number of colors contained in each list is also maintained and is used in the next step. Figure 3.3 illustrates a sample sort array.

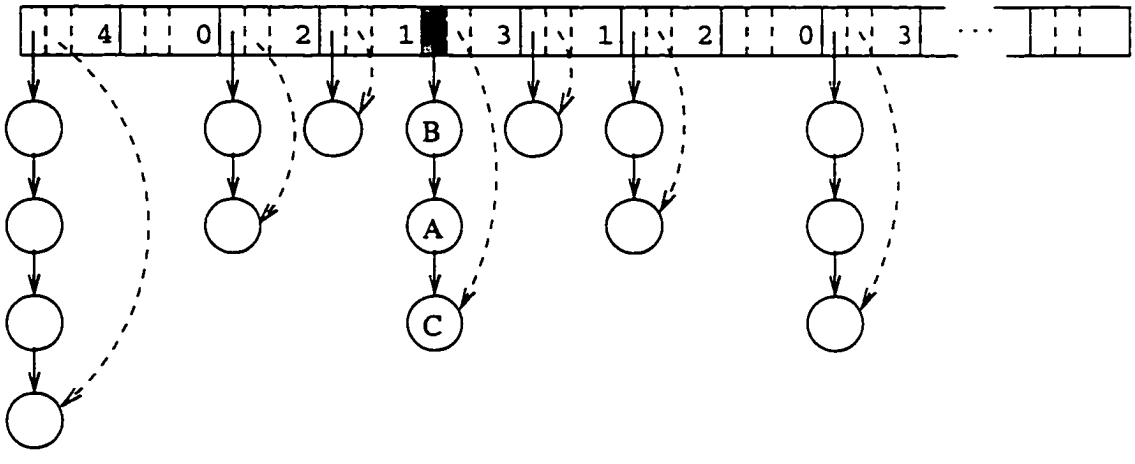


Figure 3.3: The array of singly-linked lists of colors. The colors in the current box are sorted by first separating them into lists based on the primary sort key. The solid pointers represent the links and the head pointer; the dashed lines represent the tail pointers.

The result of creating the sort array is an ordering of the colors based solely on the primary key. Sorting on the other two keys is only important for those colors that are possible candidates as the median color. One of the linked lists, M, in the sort array will contain the median color. Only the items in M need to be sorted using the second and third keys. All the colors in the linked lists preceding list M will precede the median color in a

completely sorted list. Likewise, the colors in the lists following M will follow the median color.

Therefore, the first step is to find the linked list in the sort array that contains the median color. This is easily handled by traversing the array and tallying the number of colors in each list until the list containing the median color is found. After the list containing the median color is found, it is sorted using the other two keys. This sorting can be done using the same sort array structure as the original sort.

After sorting the median list, the original list L is split into the two sub lists L_1 and L_2. L_1 contains all the colors in the linked lists preceding list M, while L_2 contains all the colors following list M. These lists are constructed by linking all the smaller lists. The time required for this process is reduced by maintaining a tail pointer in each element of the sort array.

Next, list M is split by adding one set of the colors to L_1 and the other set to L_2. If L contains an even number of colors, the split is made so that L_1 and L_2 contain the same number of colors. For an odd number of colors, the split is made such that L_2 contains one more color than L_1. Figure 3.4 illustrates the connections made to the example in Figure 3.3 in order to create the two sub lists.

3.3.3 Constructing the Colormap and Partition

After creating the k sub lists, the representative set, Y, is constructed by computing the average weighted color of each sub list. The partition, P, is then constructed using the nearest representative mapping scheme.

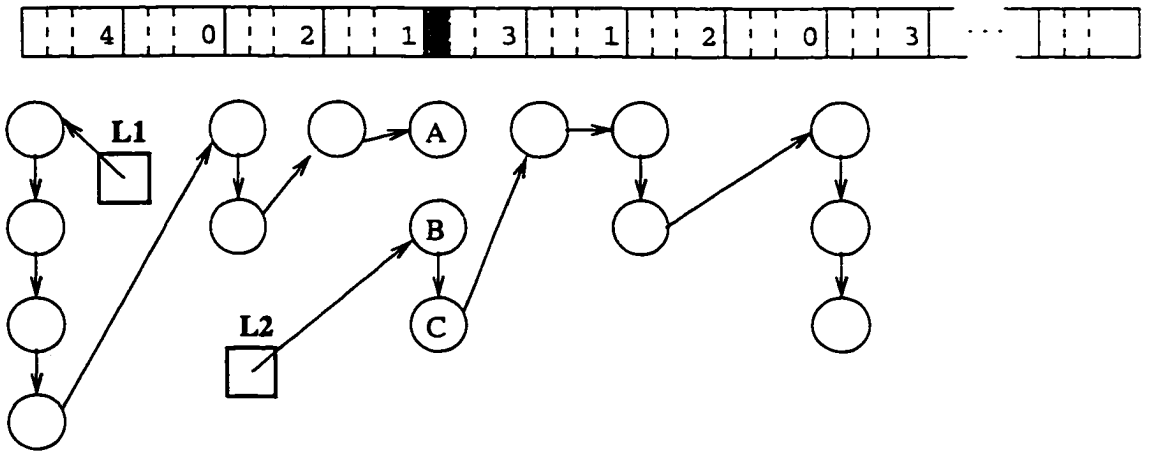


Figure 3.4: After finding the point at which to split the original list, the individual linked lists from the sort array are formed into two sub lists.

In practice, each color in the image is mapped to a representative color. This reduces the processing time since it is common for some colors to occur more than once in the image. Thus, the number of nearest neighbor searches is greatly reduced.

The mapping of original colors to a representative color is handled using the histogram which is still intact. The actual representative color is not stored. Instead, the index of the representative color is stored in the count field of the various nodes. A search is performed for a pixel's color in the histogram when the image is recolored. The representative index is retrieved from the count field and the pixel is recolored with the corresponding representative color.

To reduce the total execution time, the centroid mapping scheme can be used instead of the nearest representative mapping scheme to construct the partition, P. This scheme does not produce especially good results, but the results are better than those produced by the other common algorithms. The centroid mapping scheme can be easily implemented with the existing data structures. The mapping of the original colors to a representative color is

handled by traversing the linked list of each cluster. During this traversal, the count field is set to the colormap index of the cluster's representative color.

3.3.4 Color Reduction

To reduce the number of colors that need to be processed, many of the heuristic algorithms perform a bit reduction by setting a number of the least significant bits of each color component to zero. The common reason for this reduction is the lack of computer memory. If the color cut algorithm is to be used on a computer with limited memory, a bit reduction can be performed with few modifications to the data structures and routines. The modifications described in this section can be used with any bit reduction scheme.

The main modification is with the construction of the histogram. The same array and tree structures are used but with modified tree nodes. A sixth field is added to the node data structure called the tally field which is an one-dimensional integer array of three elements. To construct the histogram, both the original color and the reduced color (created using some bit reduction scheme similar to those described earlier in this chapter) are used in the tree insertion. The binary tree into which the color is to be inserted is found using the reduced color. The reduced color is also used to find the position within the binary tree.

When adding a new node in a binary tree, the color field is set to the reduced color while the tally field is initialized to the original color. When accessing a node for a reduced color that is already in the histogram, the tally field is incremented by the original color and the count field is incremented by one.

The second modification is actually the inclusion of an additional step. Instead of using the reduced color to sort and split the list, a new color is computed for each node in the

list. The new color for a given node, which is stored in the color field, is simply the average color computed by dividing (and rounding to the nearest integer) the tally field by the count field.

The final modification required to implement the color reduction is made in the creation of the representative color set. Instead of summing the colors in each list using the color fields, the tally fields are used. This produces a better representative color since the original colors are considered and not just the reduced colors.

3.4 Algorithm Analysis

3.4.1 Worst Case Analysis

In evaluating the color cut quantizer algorithm, I derive the time complexity for the worst case running time. This is the method used by the authors of the heuristic quantization algorithms reviewed earlier.

Theorem 3.1 *A quantizer Q with a representative set of k colors can be constructed by the color cut quantization algorithm for an image containing n pixels in $O(n \log k)$ time, in the worst case.*

The proof of Theorem 3.1 is derived from the following lemmas.

Lemma 3.1 *The histogram used by the color cut quantization algorithm can be constructed for an image containing n pixels with a worst case time of $O(n)$.*

Proof: The algorithm builds the histogram by traversing the image and inserting the color of each pixel into the histogram data structure. If the image contains n pixels, then in the

worst case, each pixel contains a unique color and thus there are n colors.

The insertion of a single color into the histogram involves the insertion of a color into a binary tree rooted at one of the 256×256 array elements and the insertion into a linked list. Finding the array element is straight forward and requires constant time. Likewise, the insertion of the color into the linked list requires constant time since the color is inserted at the front of the list.

The only time actually consumed in the process is in the insertion of the color into the proper binary tree. The worst case analysis for the insertion of an item into a binary tree occurs when the nodes form an ordered singly-linked list. The insertion of an item into this ordered list requires $O(x)$ time where x is the number of items in the list. Since all nodes in the binary tree have the same red and green components, they only differ in their blue components. Thus, the maximum number of nodes in a binary tree in the histogram structure is 256. Hence, in the worst case, an insertion of a single item into the histogram requires $O(1)$ time. The algorithm repeats this insertion process for each of the n colors and results in $O(n)$ time, in the worst case. ■

Lemma 3.2 *Given a singly linked list of length n, the initial partition containing k clusters can be constructed by the color cut algorithm with a worst case time of $O(n \log k)$.*

Proof: The construction of the partition involves a repetitive subdivision of the colors in a linked list into smaller and smaller lists of approximately equal size. This process is repeated until k lists are created. Since the number of colors in the list varies for each iteration, it is first analyzed for a single iteration assuming a list of length v.

First, the computation of the minimum bounding box of the colors in a given list requires a single iteration over all the colors and six comparisons for each color. Hence, the construction of the bounding box requires $6v$ comparisons or $O(v)$ in the worst case. After the bounding box is constructed, the sort keys are determined based on the span of the box in each dimension. This can be done in constant time.

Given the sort keys, the color list is then sorted and split at the median entry. Using the sort array as described in Section 3.3.2, the colors are split into smaller lists based on the primary key. This process takes linear time since the list must be traversed and each color inserted into one of the smaller lists. The actual insertion of an item into one of the smaller lists can be done in constant time since it is inserted at the front of the list using the head pointer.

After the original list of colors is split into the smaller lists, only the list containing the median entry is actually sorted. This list can be located in constant time. The median list will contain, at most, $v - 1$ colors since the items were divided based on the primary key which represents the largest dimension of the minimum bounding box; at least one entry must be contained in a different list. Using the my sort array structure, the median list can be sorted on the secondary key in $O(v)$ time since the median list can contain $v - 1$ colors. Again, the list containing the median color can be located in constant time. The sorting of this median list on the final key requires constant time since it can contain at most 256 colors.

After the median list is sorted, the median entry is located which requires linear time. The list is then split into two sub lists of approximately the same size. The two sub lists

can be created in $O(v)$ time since in the worst case, the median list contains all but one of the colors and the median entry would of course be in the middle of the list.

Combining the worst case time complexities for a single iteration of the partition construction yields a worst case time of

$$T(v) = O(v).$$

The evaluation of the actual partition construction can be done using a recurrence relation since each of the k iterations reduces the size of the given list by half. The time function for this recurrence is derived from the time for a single iteration,

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

where $T(\frac{n}{k}) = O(1)$. For simplicity, assume that k is a power of 2. From the construction of a recursion tree, we find a worst case complexity of

$$T(n) = O(n \log k).$$

■

Lemma 3.3 *Given k singly linked lists, the union of which contains n items, the color cut algorithm can construct a colormap containing k entries in $O(n)$, in the worst case.*

Proof: The construction of the colormap involves a single traversal through each of the k sub lists. Since the sum of the lengths of all k sub lists is n, the worst case for the colormap construction is $O(n)$. ■

Proof of Theorem 3.1: The color cut quantization algorithm can be divided into three parts: building the histogram, constructing the partition and constructing the colormap. Lemma 3.1 shows that the histogram can be constructed in $O(n)$ time in the worst case. The construction of the partition has a worst case of $O(n \log k)$, from Lemma 3.2. From Lemma 3.3, the colormap can be constructed in linear time. Hence, the worst case time complexity of the color cut algorithm is $O(n \log k)$. ■

The worst case time complexity of the median cut and center cut algorithms [35] is $T_1(n) = O(c \log k)$ where c is the number of colors after performing the color reduction. The octree quantization algorithm [29] has a worst case time complexity of $T_2(n) = O(n)$ when no preference is given to the selection of the node to be reduced in the reduction step. If some preference is given, the worst case is $T_3(n) = O(nk)$.

The analytical analysis performed in this section shows that the color cut quantization algorithm is no worse than that of the other popular algorithms. A better analysis would be based on the average time complexity. However, that would require knowing what an average input set was or using random input sets. But images are not just a random set of values. Thus, the computation of the average time complexity is not realistic.

3.4.2 Execution Analysis

To verify that the actual execution time of the color cut algorithm is acceptable when compared to the other popular algorithms, I performed several execution time experiments[2]. The experiments were performed for several of the quantization algorithms using the set of

[2]The experiments were performed on an Intel (tm) Pentium (tm) P5 100Mhz processor based machine running Linux (tm) and X Windows (tm) with 64M of physical memory. The machine was connected to a network but all files (both system and data) were on a local drive.

test images described in Appendix C. The elapsed time for each algorithm[3], was computed from the point after the image was loaded up to the point when the color map was created. The image was loaded into memory and stored in an array.

The first experiment compared the execution times between the color cut and octree quantization algorithms. Both of these algorithms work with the full set of original colors. The results are illustrated in the graph of Figure 3.5

For the second experiment, I compared the execution times between the median cut algorithm and the color cut algorithm with a 3-3-3 bit reduction. Thus, each algorithm handled the same reduced set of colors. The results from this experiment are illustrated in Figure 3.6.

3.5 Algorithm Comparisons

To test the quality of the color cut algorithm, I performed several experiments on the set of test images described in Appendix C. These experiments involved quantizing images by recoloring and evaluating the quantization errors.

3.5.1 The Color Cut Algorithm

For the first experiment, I quantized each of 50 images using the color cut algorithm (with nearest representative mapping) and the three most popular heuristic algorithms – the median cut, center cut, and octree algorithms. A value of $k = 256$ was used in the design of each quantizer.

[3]Kruger's [45] implementation of the median cut algorithm and Gervatz's [29] implementation of the octree algorithm were used for the experiments. In the octree algorithm, the next leaf node selected to be reduced was the one representing the least number of pixels.

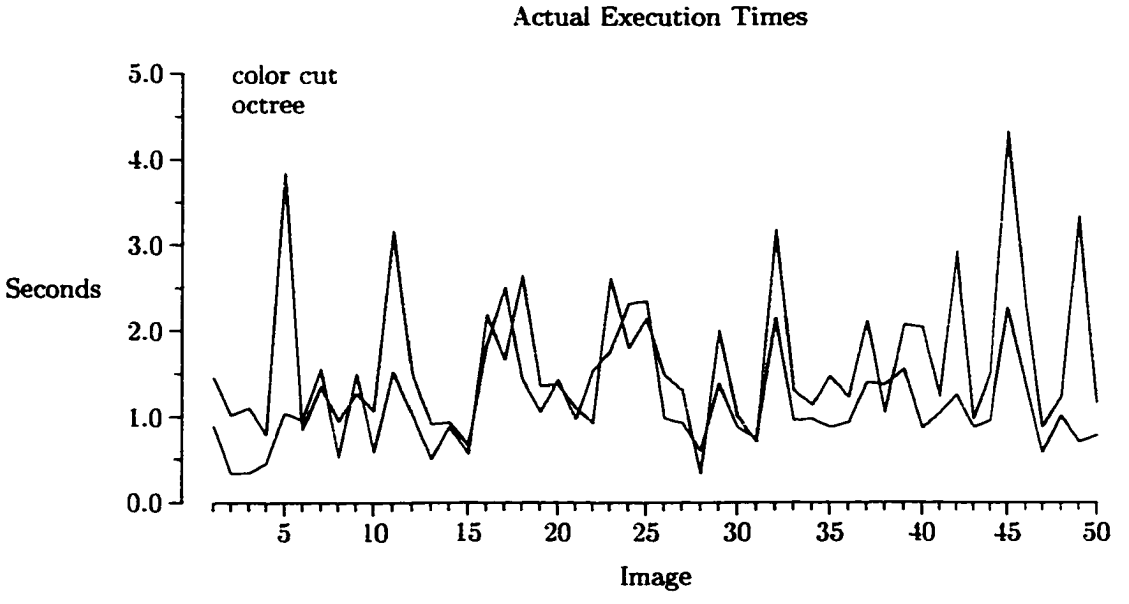


Figure 3.5: A comparison of the actual running times between the color cut (with no bit reduction) and the octree quantization algorithms.

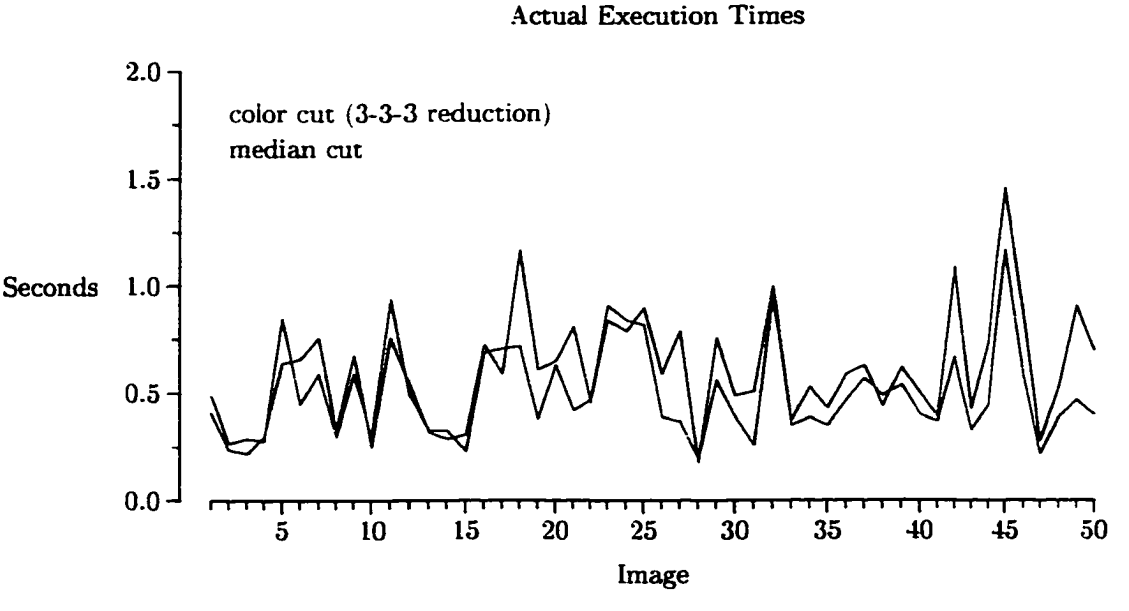


Figure 3.6: A comparison of the actual running times between the color cut algorithm with a 3-3-3 bit reduction and the median cut algorithms.

The quantization error was then measured for each of the resulting quantized images using the RMS metric. The results are illustrated by the graph in Figure 3.7. The values from each algorithm are illustrated by a different color. The images, numbered 1 to 50 across the horizontal axis, are ordered by increasing RMS error values based on the color cut algorithm.

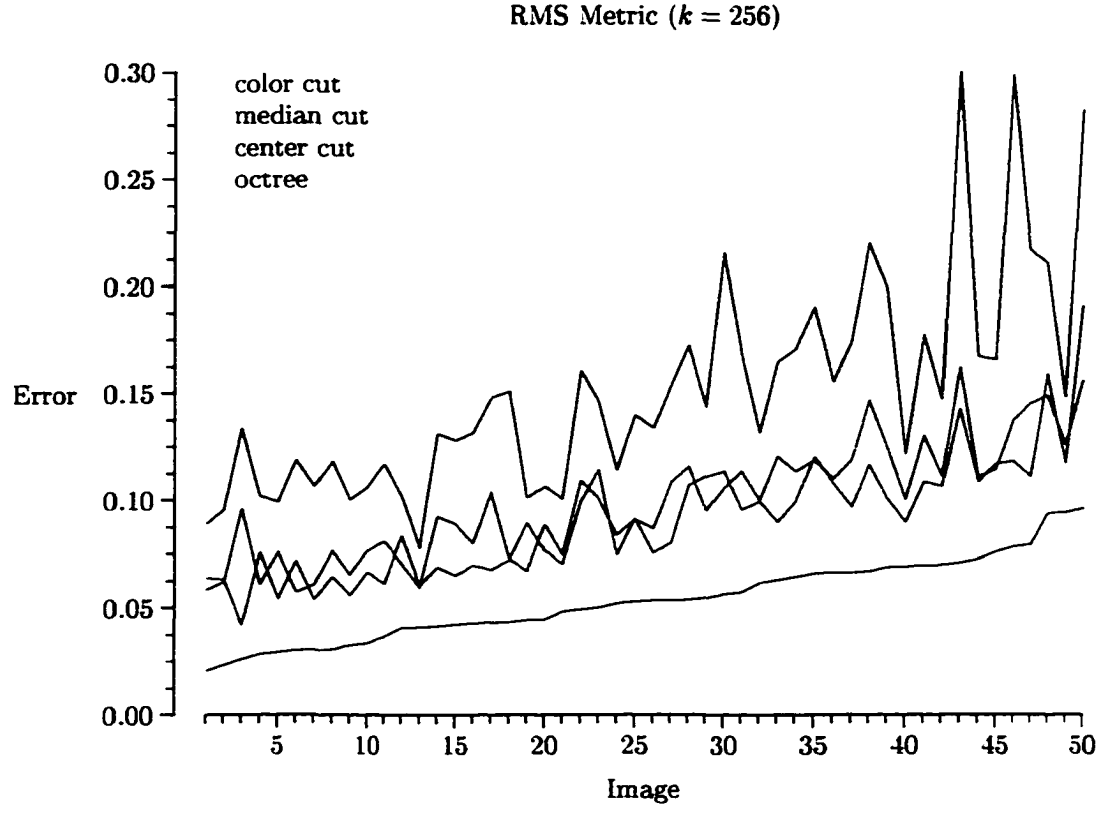


Figure 3.7: Algorithm comparisons using the RMS metric. The images are ordered by increasing relative RMS error values based on the color cut algorithm.

From these results and from visual inspection, it appears that the color cut algorithm produces better quantized results for all images than the other three algorithms. The color cut algorithm produced better quantized images, for all 50 images, than any of the other three algorithms.

I also measured the quantization error using the $CRMS_g$ metric to further verify its usefulness as a quantization error metric. The results are illustrated by the graph in Figure 3.8 with the images ordered as in Figure 3.7. The $CRMS_g$ metric results seem similar to those produced by the RMS metric. This is especially true for the color cut algorithm.

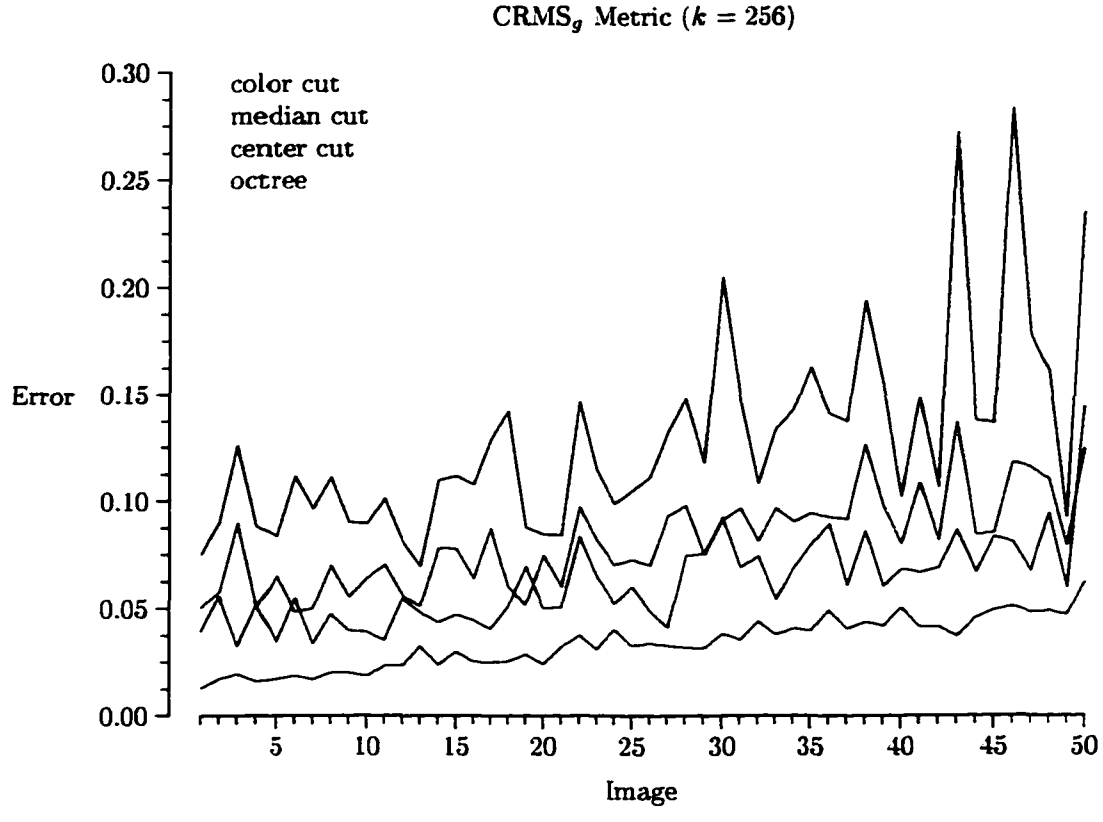


Figure 3.8: Algorithm comparisons using the $CRMS_g$ metric. This is the same comparison as in Figure 3.7 but using the $CRMS_g$ metric.

3.5.2 Color Cut with Centroid Mapping

Next, I conducted an experiment to see how the color cut algorithm performs when using the centroid mapping scheme instead of the nearest representative scheme. In this experiment, I quantized the set of test images using the color cut algorithm as before but with centroid

mapping. Then I compared it to the other three popular algorithms as shown in Figures 3.9 and 3.10. In all cases except one, using the RMS and $CRMS_g$ metric, the color cut algorithm with centroid mapping did better than the other algorithms. In one case using the RMS metric, the result was the same as for the median cut and octree quantization algorithms.

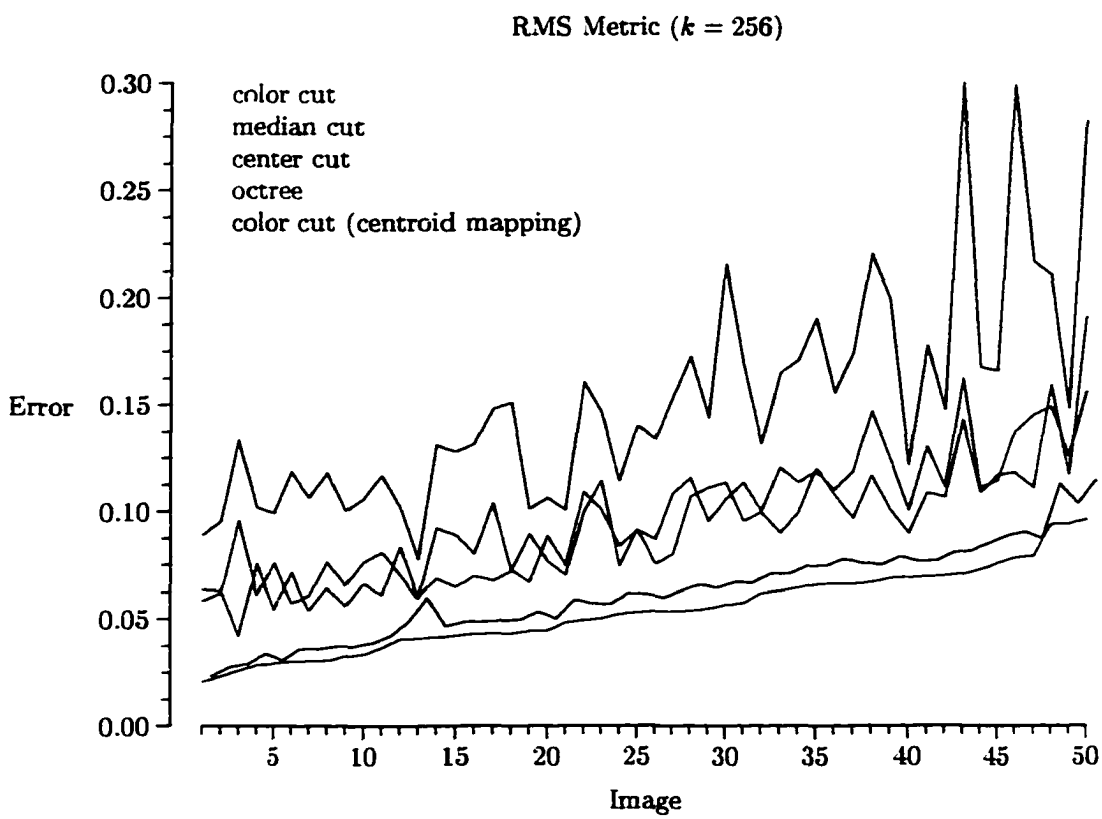


Figure 3.9: A comparison of the color cut algorithm using centroid mapping based on the RMS metric. The comparison is made with the three popular algorithms and the original color cut algorithm with nearest-representative mapping.

3.5.3 Color Cut with Color Reduction

Finally, I conducted an experiment to see how well the color cut algorithm performs when using the color reduction scheme described in Section 3.3.4. For this experiment, the color

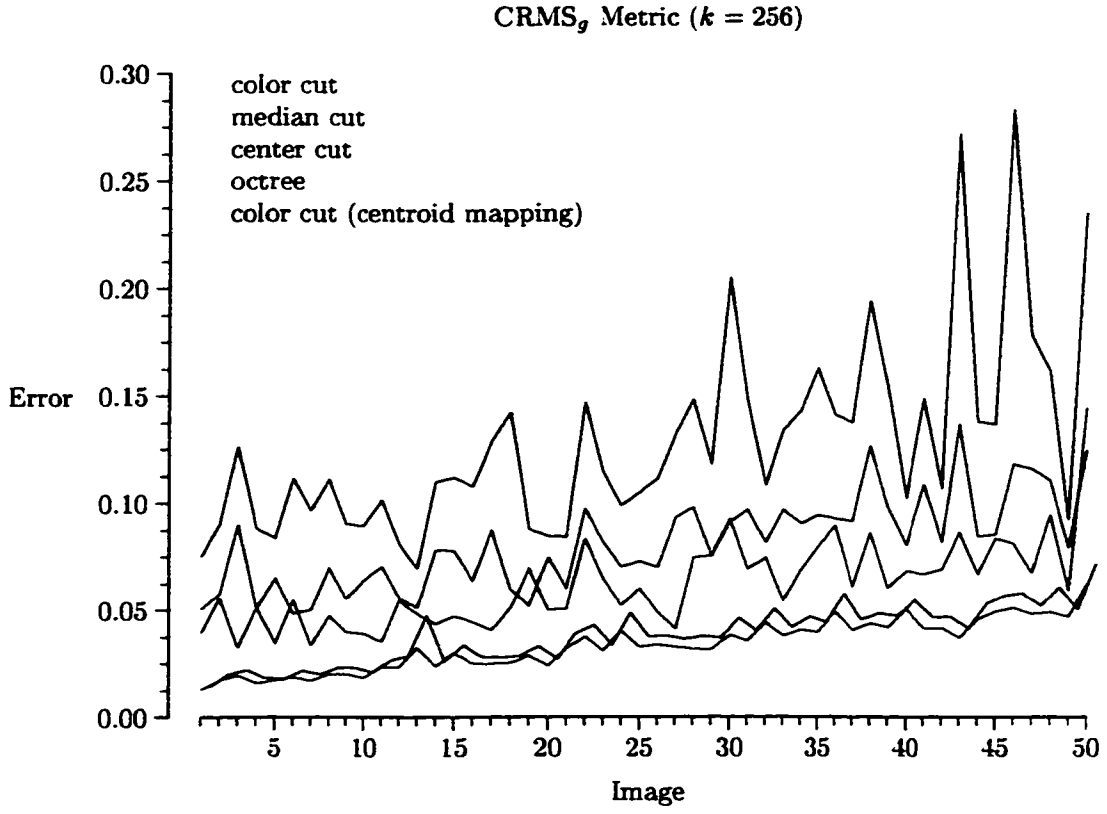


Figure 3.10: Color cut with centroid mapping (CRMS$_g$ metric). This is the same comparison as in Figure 3.9 using the LARMS metric.

cut algorithm was used but with three different bit-reduction schemes. The three schemes consisted of chopping off 1-, 2-, and 3-bits, respectively, with each color component being reduced by the same number of bits. I then compared the quantization error results from the CRMS$_g$ metric with those of the color cut algorithm with no bit reduction. The results are illustrated by the graphs in Figures 3.11 through 3.13. I also compared the results between the 3-3-3 reduction scheme and the median cut and octree algorithms. The results of this experiment are illustrated in Figure 3.14.

Reducing the colors by one-bit does not appear to make a significant visual difference in the results. In fact, there is only a marginal difference between the one-bit reduction

and the results with no reduction. The larger bit reductions, on the other hand, produced larger error values and corresponding visual discrepancies.

From these experiments, based on the RMS and $CRMS_g$ metrics, as well as visual inspection, it appears that the color cut algorithm produces better results than the other three common algorithms. In the case of the color cut algorithm with bit-reduction, it appears that the consideration of the original colors (before reduction) in the computation of the representative values does make a difference. Especially since the median cut algorithm uses a 3-3-3 bit reduction and the same reduction with the color cut algorithm still produced far better results.

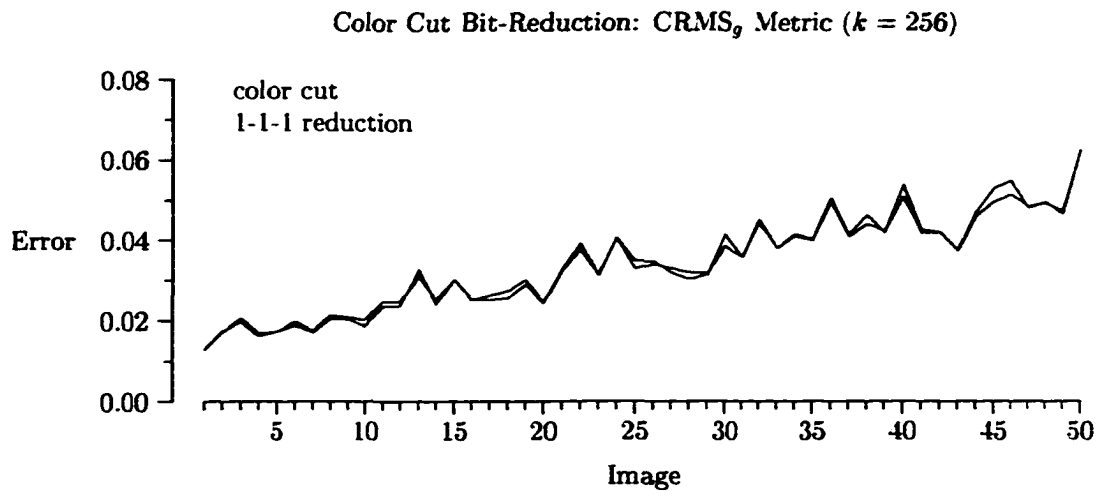


Figure 3.11: A comparison of the color cut algorithm between the original with no color reduction and the use of a 1-1-1 bit reduction scheme.

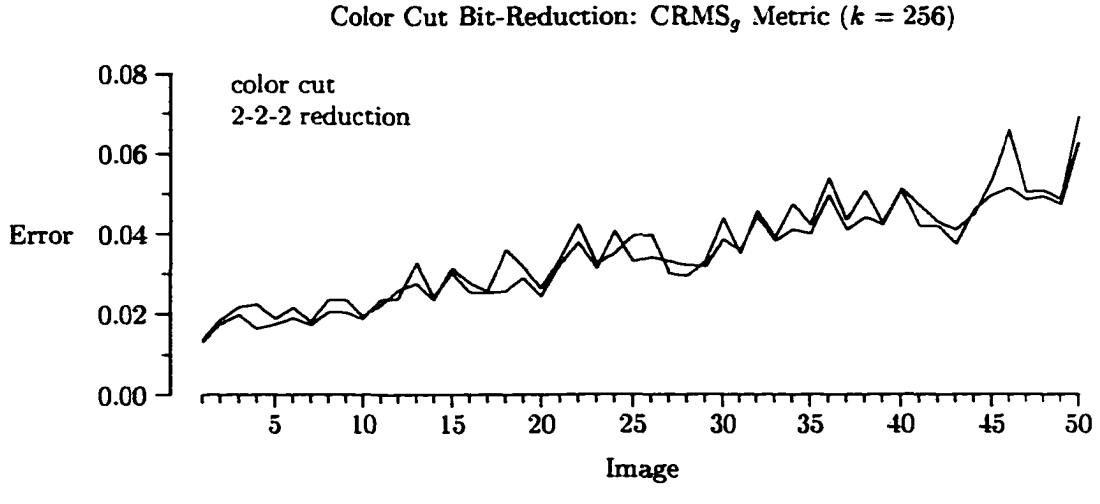


Figure 3.12: A comparison of the color cut algorithm between the original with no color reduction and the use of a 2-2-2 bit reduction scheme.

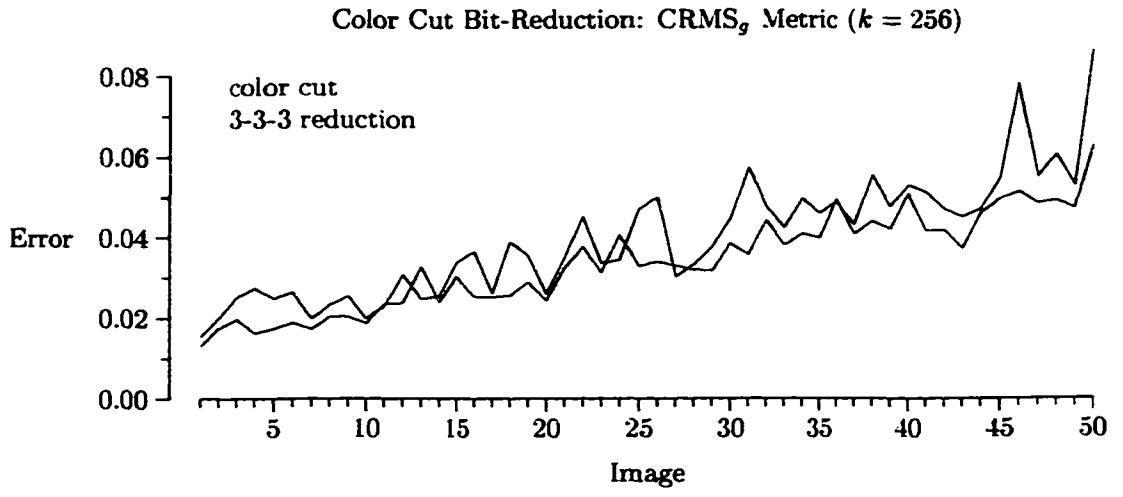


Figure 3.13: A comparison of the color cut algorithm between the original with no color reduction and the use of a 3-3-3 bit reduction scheme.

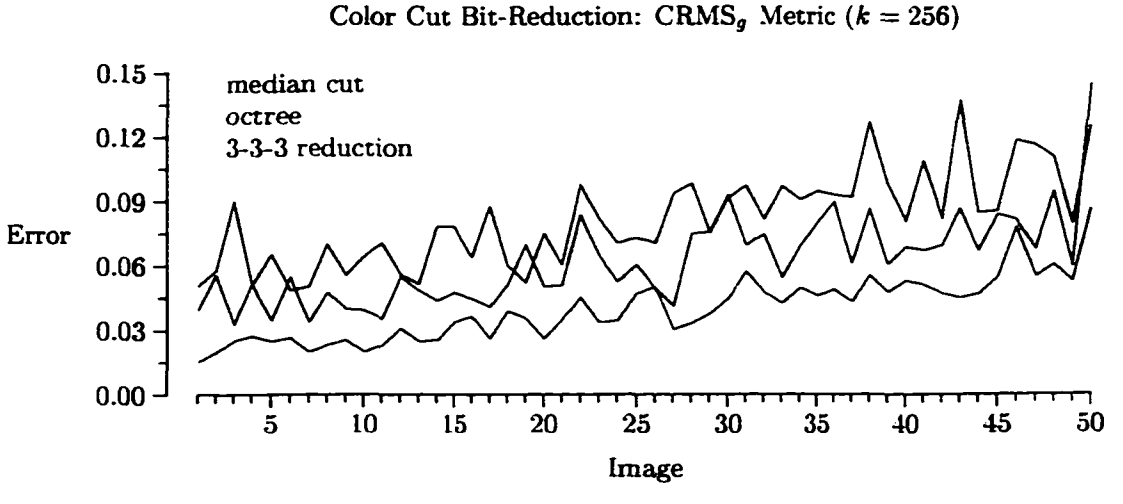


Figure 3.14: A comparison between the color cut algorithm with a 3-3-3 bit reduction scheme and the median cut and octree algorithms.

Chapter 4

Partitioning by Nearest-Neighbor

4.1 Introduction

The most common type of heuristic color quantizer is one in which the color space is partitioned such that a query color maps to the nearest representative color. Nearest representative color mapping is simply the classic nearest-neighbor search problem. The straight forward approach is to use an exhaustive search in which the input value is tested against each representative to find the one that minimizes the Euclidean distance. This method, however, is very slow since a lot of time is spent comparing the input value to representatives that could not possibly be the nearest-neighbor. Other solutions have been proposed for the solution of the nearest-neighbor problem [79, 86]. For use in the quantization process, the most popular solution is the locally sorted search algorithm due to Heckbert [35]. In this chapter, I review Heckbert's method and then introduce a new search technique which results in an experimentally better search time compared with the locally sorted search algorithm.

4.2 Locally Sorted Search

The locally sorted search (LSS) algorithm uses a preprocessing step to limit the number of representative colors that must be searched for each query color. First, the RGB color space is divided into equal sized cubes. Associated with each of these cubes, is a list of only those representative colors that could possibly be a nearest-neighbor of a color within that cube. In the search for a nearest-neighbor, we determine which cube contains the original color and perform an exhaustive search on the list of possible neighbors. This technique works well because it eliminates many of the colors from possible consideration in the nearest-neighbor search process.

The list of colors associated with each cube is limited by determining the smallest area around the cube which could contain a nearest-neighbor. Consider Figure 4.1 which illustrates a 2-D version of the problem.

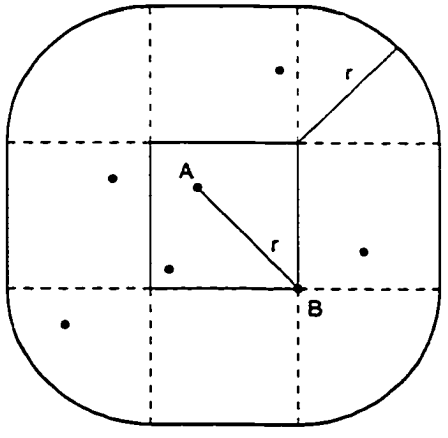


Figure 4.1: The minimum area around a cube (solid line) that must be searched for a possible nearest neighbor. The dots show the locations of representative colors [35].

To build the list of colors for each cube, the representative colors are searched to find the color (A) that is closest to the center of the cube. Next, the vertex (B) most distant

from this point is determined and the distance (r) is calculated. For another representative color to be the nearest-neighbor of a color within the cube, it must lie within r distance from the cube. Any representative color within this area, as illustrated in Figure 4.1, may be a nearest-neighbor of a color within the cube, while any color outside can not be. The list of colors associated with each cube are those colors lying within r distance from the cube. The colors in the list are sorted based upon their distance from the edge of the cube; those colors inside the box have a distance of zero. To reduce the time required to build the list for each cube, the lists are only created for those cubes that are actually referenced.

Heckbert's LSS algorithm reduces the number of representative colors that must be searched when mapping a color. However, there is still a drawback to this approach. The division of the RGB color space into equal sized cubes leave many of the cubes unused. This leads to there being some lists containing a large number of eligible nearest-neighbors. Some of the boxes actually referenced will contain many colors while others may not contain any. In the former case, all of the representative colors within the cube must be placed on the list in addition to others within the search range. In those cases where no representative colors lie inside the cube, the search area will be very large with a good chance of a large number of possible nearest-neighbors.

4.3 Adaptive Locally Sorted Search

To improve the search time for a nearest-neighbor, I developed the Adaptive Locally Sorted Search (ALSS) algorithm which is based on the LSS algorithm. In the ALSS algorithm, I subdivide the RGB color space into boxes of varying sizes instead of cubes and create

a possible nearest-neighbor search list for each box in a similar manner to that of the LSS algorithm. This allows us to adapt the boxes and search lists to the colors in the representative color map. In the sparse areas of the color space, the boxes will be large, while in the denser areas, they will be small. In all cases, the lists tend to be about as short as possible.

We limit the number of boxes created such that each box will contain only one representative color. Thus, we can create all of the search lists in a preprocessing phase since there is a high likelihood that every representative color will be used.

The subdivision of the color space is handled using an adaptive k-d tree [21, 86]. We repeatedly split the color space along one of the three dimensions until k boxes are created, where k is the number of colors in the representative color map. The interior nodes of the tree contain information about the split while the leaf nodes contain the representative color and the list of possible nearest-neighbors. The structures used to represent the nodes of the k-d tree are shown below

```
struct ListItem {
   rgb color;
   int minDist;
};

struct kdnode {
   byte leaf;  // Is this a leaf node?
   byte split; // Value of split split.
   byte index; // Component of the split.
   kdnode *left,
          *right;
   rgb      *color; // Representative color.
   ListItem *list;  // The search list.
   int       size;  // Size of the search list.
};
```

4.3.1 Construction of the k-d Tree

The construction of the k-d tree begins with a list (O) of the k colors in the representative color map. From this list, the minimum bounding box is computed and the longest dimension is determined. If more than one dimension is the longest, the tie is broken by giving the green component the highest priority followed by red and then blue. This ordering corresponds to the natural property of the human eye in which we perceive smaller changes in green colors better than red and changes in red colors better than blue [33]. The colors in L are then sorted using the longest-dimension component as the sort key.

Next, the list is split in the middle creating two new lists, $L1$ and $L2$. The mid point along the split axis between the middle item and the next item becomes the discriminator. After this split, $L1$ contains all the colors whose component values are less than the discriminator along the split axis and $L2$ all those greater than or equal. This list splitting operation is illustrated by the following C++ code

```
int SplitList( rgb *L, int d, rgb *&L1, rgb *&L2 )
{
   int l = length( L );
   int h = l / 2;

   L1 = new rgb[ h ];
   L2 = new rgb[ l - h ];

   if( h == 1 ) h = 0;
   for( int i = 0; i < h; i++ ) L1[i] = L[i];
   for( int i = h+1, j = 0; i < l; i++, j++ ) L2[i] = L[j];

   rgb a = L[h];
   rgb b = L[h+1];
   switch( d ) {
      case 0: return( (a.r + b.r) / 2 );
      case 1: return( (a.g + b.g) / 2 );
      case 2: return( (a.b + b.b) / 2 );
   }
}
```

It is hoped that a balanced k-d tree is achieved but there are no guarantees. Thus, special care must be given in those cases where the middle item and the previous item have the same component value and where all the items have the same component value. One option is to shift the split point (originally the mid point) left or right a few places until an unique item is found. A second option is to split along a different axis.

An interior node is created to represent the split with the dimension and discriminator stored in the node. The process is then repeated for the two new lists with the resulting nodes being stored as the left and right children of the new interior node. The process ends for a given list when that list contains a single color. In this case a leaf node is created. The following C++ code illustrates the construction of the k-d tree.

```
kdnode * BuildKDTree( rgb *L, rgb *O )
{
   kdnode *node;
   int dim, pos;
   rgb *L1, *L2, *N;

   if( length( L ) == 1 ) {
      N = GetNearestNeighborList( L[0], O );
      node = AllocateNode( N );
   }
   else {
      dim = CompLongestDim( L );
      SortListByDim( L, dim );
      pos = SplitList( L, dim, L1, L2 );
      node = AllocateNode( dim, pos );
      node->left = BuildKDTree( L1, O );
      node->right = BuildKDTree( L2, O );
   }
   return( node );
}
```

After the k-d tree is constructed, the color space will be divided into k boxes of varying sizes with a single representative color lying on or within each box. Figure 4.2 illustrates the subdivision of a sample 2-D color space.

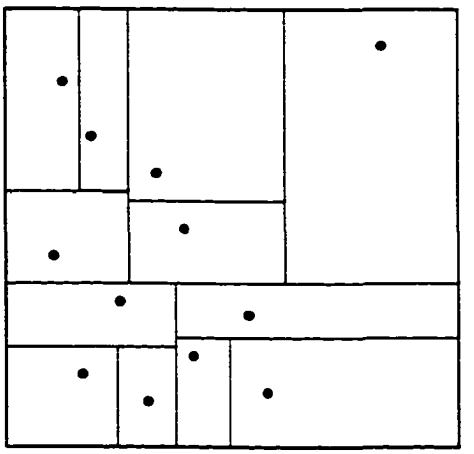


Figure 4.2: Subdivision of sample 2-D color space resulting from the construction of a kd-tree. The dots illustrate the representative colors.

After constructing the k-d tree, the list of possible nearest-neighbors for each box must be created. The boxes created by the construction of the k-d tree each contain a single representative color. Since the representative colors are stored in the leaf nodes, only the data in the leaf nodes need to be accessed to determine the size of the box containing the given representative color and create the list of possible nearest-neighbors.

4.3.2 Nearest-Neighbor Search List

The creation of the nearest-neighbor search list is similar to that of the LSS algorithm. I add an additional step, however, which reduces the number of possible nearest-neighbors more than could be done by the LSS algorithm. The ALSS algorithm requires the minimum (lower front left vertex) and maximum (upper back right vertex) vertices of the box represented by the leaf node. This information can be computed during the traversal and creation of the k-d tree. Therefore, we simply use the `BoxCoords` routine to retrieve the coordinates of the box with the assumption that the information was computed during the construction of the tree.

After retrieving the box coordinates, the vertex (v) most distant from the representative color (y) is computed. This can be easily handled by computing the center of the box and determining in which octant the representative color lies. In the LSS algorithm, the representative color closest to the center of the box was located and used to determine the most distant vertex. Since there is only one color within our boxes, we can remove this step and reduce the preprocessing time. With v identified, the distance (r) between y and v is calculated. We expand the size of the original box by r, as illustrated in Figure 4.3, and term this new box the *search box*. The search box represents the region within which a color must lie to be a candidate as a nearest-neighbor to a color within the original box.

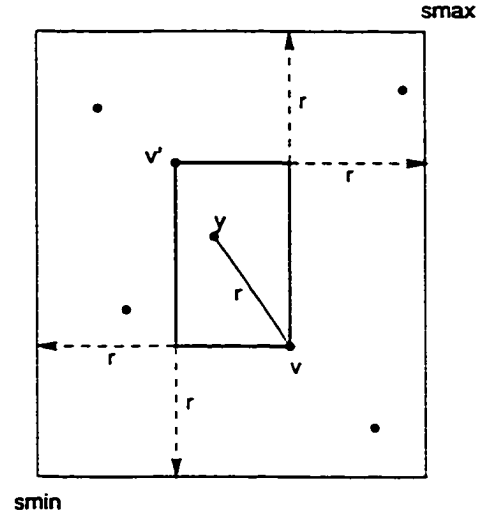


Figure 4.3: The result of expanding a box in the ALSS algorithm to include all colors that are possible nearest-neighbors to a color in the original box (dark lines).

After creating the search box, we create a list (N) of the colors from the original list of representatives (O) which lie entirely within the box. The representative color within the original box is not added to the list at this time. The following routine illustrates the construction of the nearest-neighbor search list for a given box. The routine `inbox()`

determines if a given representative color is within the search box. The search box is identified by its lower-left and upper right vertices.

```
list GetNearestNeighborList( rgb y, rgb *O )
{
  rgb  v, bmin, bmax, smin, smax;

  BoxCoords( bmin, bmax );
  v = DistantVertex( y, bmin, bmax );

  int d = squareRoot( y, v );
  smin = bmin - d;    // Component by component.
  smax = bmax + d;    // Component by component.

     /* Get rep colors that lie inside the search box. */
  rgb *N = new rgb[ length( O ) ];
  for( int i = 0, j = 0; i < length( O ); i++ )
    if( O[i] != y && inbox( O[i], smin, smax ) )
      N[j++] = O[i];

  N = VertexReduction( bmin, bmax, N, y );
  SortTheList( N, bmin, bmax );
  return( N );
}
```

While list N contains all the possible nearest-neighbors of a color lying within the original box, we observe that in some cases, this list contains colors that could not possibly be the nearest-neighbor of any color within the original box. Therefore, I attempt to reduce the size of list N by removing any additional representatives that can not possibly be a nearest-neighbor of a color within the original box. Consider the vertex v' of the original box in Figure 4.3. For a color in N to be its nearest-neighbor, that color must be closer to v' than to y. The region in which that color must lie is a sphere centered at v' whose radius is the distance from y to v' as illustrated in Figure 4.4. Since no representative color in our example lies within the sphere centered at v', then y is closer to v' than any other representative color.

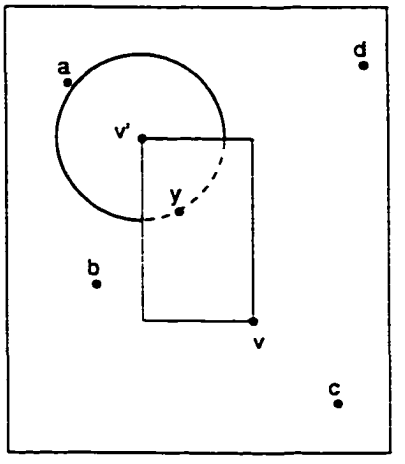


Figure 4.4: The area in which a nearest-neighbor of a vertex, v' must lie is the spherical region where v' is the center and the distance from v' to y is the radius.

If we created a sphere for each discrete color point in the original box, in the same manner as that for v', we could determine each representative color that is a nearest-neighbor of any color within the original box. This can be done by determining if any color in N lies within one of the spheres. Those representative colors that are not within any of the spheres can be removed from N.

Checking each representative color in N against a sphere centered at each discrete point within the original box would be time consuming. Thus, I make the observation that we do not need to check a representative color against every discrete point within the original box. Instead, we need only check the colors in N to determine if they lie within a single sphere centered at one of the vertices. The proof of this observation is provided in Appendix A.

To perform this evaluation, we first create a sphere centered at each vertex whose radius is the distance from the respective vertex to y. Next, the original box is subdivided into octants with the split being made at the representative color y. The result of performing these steps on our 2D example is illustrated in Figure 4.5.

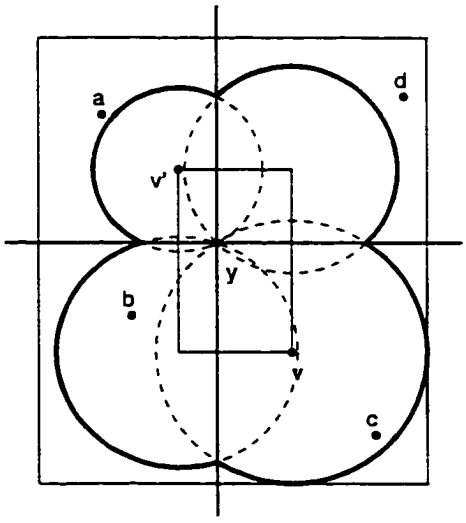


Figure 4.5: Placing a sphere at each vertex of the cube creates the minimum area (dark lines) in which a color must lie to be a nearest-neighbor of a color in the given box.

To determine if a color needs to be removed from the search list, it is determined in which octant the color lies. The color is then evaluated to determine if it lies within the sphere centered at the vertex of the box contained in the given octant. If the color is within the sphere, it remains a part of the search list, otherwise, it is removed. In our example, colors a and d would be removed, while b and c remain. We conclude this vertex reduction process by adding the representative color to the search list and sorting the list of colors in ascending order by their distance from the edge of the box; the distance of the representative color is zero since it lies within the box.

4.3.3 Nearest-Neighbor Search

When searching for the closest representative color in the k-d tree, the tree is traversed in the normal manner until a leaf node is reached. Upon reaching a leaf node, we use the information computed in the preprocessing phase to determine which representative

color is the nearest-neighbor of the given color, p, for which the search was made. The nearest-neighbor of p will either be the representative color of the box or one of the colors within the nearest-neighbor list. To find the nearest-neighbor of a given color, x, we find the color within the list of possible nearest-neighbors that is closest to x. We do not have to actually traverse the entire list of possible colors. Instead, we can stop the traversal when the distance between x and a color in the sorted list is less than the distance between the next color in the list and the edge of the search box. This test is easily made since we stored the distance between a color and the edge of the box in the color list. The following function describes this search process.

```
rgb NearestNeighbor( rgb x, rgb *O )
{
   point y;
   int dist, nearest;
   int i = 0, min = ∞;

   while( min > edgedist( O[i] ) ) {
      y = O[i];
      dist = squareRoot( y, x );
      if( dist < min ) {
         nearest = i;
         min = dist;
      }
      i++;
   }
   return( O[nearest] );
}
```

4.4 Algorithm Analysis

To evaluate the effectiveness of the ALSS algorithm, I examine the worst case analysis for the construction of the boxes and search lists. This analysis is then compared to that of

the LSS algorithm. Finally, I provide a comparison between the two algorithms based upon empirical tests performed on a number of images.

4.4.1 Worst Case Analysis

Heckbert found that his LSS algorithm required $O(d^3k + d^3L \log L)$ time in the worst case to construct the search lists, where d is the number of cell divisions along each dimension and L is the average list length. The subdivision of the color space into equal sized cells requires $O(1)$ time since a 3-D array is used and no computations are required to split the color space.

Theorem 4.1 *Given a list of k representative colors, the worst case time for the construction of the k search lists by the ALSS algorithm is* $O(k^2)$.

Proof: The evaluation of the worst case time complexity for the ALSS algorithm follows that of the LSS algorithm. First, I examine the worst case for the construction of a single search list. The first step in the construction process is the evaluation of the k representative colors to determine if they lie within the search area. This operation requires $O(k)$ time since every representative color must be evaluated. Next, the vertex reduction step is performed in which each color in the search list must be evaluated against one of the vertices. This step requires $O(k)$ time as does the distance computations. Finally, the sorting of the colors by their distance from the edge of the original box requires $O(k)$ time using a radix sort. Thus, the construction of a single search list requires $O(k)$ time in the worst case while the construction of all k lists requires $O(k^2)$. The subdivision of the color space into k boxes of varying sizes using the k-d tree structure requires $O(k^2)$ time in the worst case since the worst case result would be a linear list. ■

Theorem 4.2 *The time to search for a nearest-neighbor from a list of k colors using the ALSS algorithm is $O(k)$ in the worst case with a preprocessing time of $O(k^2)$.*

Proof: The search list of the ALSS algorithm can be constructed in time $O(k^2)$ in the worst case based on Theorem 4.1. When searching for a nearest representative color, the kd-tree must be traversed to find the search box containing the given color. This requires $O(k)$ in the worst case since the tree could be a linear list. Once the box is found, the nearest representative color from the list of potential candidates must be searched. In the worst case, the search list will contain all k colors. Thus, $O(k)$ time is required in the worst case. Combining the times for these two steps results in a worst case time for finding a nearest-neighbor using the ALSS algorithm of $O(k)$. ■

Heckbert showed that his LSS algorithm also required $O(k)$ time to search for a nearest-neighbor. If the image contains n, pixels, the total time to locate every nearest representative color requires $O(nk)$ time in the worst case for both the LSS and ALSS algorithms.

4.4.2 Empirical Analysis

Even though the worst case search times are equal for the two algorithms, experimentation suggests that, in practical terms, my algorithm performs better. For my experiment, I used the 50 images described in Appendix C. A colormap of 256 colors was generated for each image using the color cut quantization method. The LSS and ALSS algorithm were each used to construct the appropriate search lists and to compute empirical data. I used a value of $d = 8$ for the cell division of the LSS algorithm. For the LSS algorithm, I used the code by John Bradley [9] for the implementation in his image manipulation program.

	LSS	ALSS
Average minimum list size	5.90	3.26
Average maximum list size	255.18	218.54
Average list size	89.09	63.25
Average number of lists	222.62	256.00
Average number tested	16.29	6.06

Figure 4.6: Experimental results comparing the LSS and ALSS algorithms.

From my experiments, the results of which are provided in Figure 4.6, I found that the average search list size was 89.09 for the LSS algorithm and 63.25 for the ALSS algorithm. But the entire list does not have to be searched each time. So I also computed the average number of representative colors that must be tested when searching for a nearest-neighbor. For the LSS algorithm, it was 16.29 and for the ALSS algorithm it was 6.06. The test involved in searching for a nearest-neighbor includes computing the Euclidean distance between the color (for which the search was made) and a representative color. Thus, in the LSS algorithm, an average of 16.29 distance computations must be computed for each pixel in the image. But for the ALSS algorithm, it is only 6.06.

Both algorithms perform the same number of logical comparisons (three) for each distance computation required. But the LSS requires one additional computation before beginning the distance computations. On the other hand, the ALSS requires one comparison for each level in the kd-tree that must be traversed to reach a leaf node. In my experiment, I found that the average tree traversal was 8. Thus, 8 comparisons were required by the ALSS to reach the leaf node before performing the distance computations. Using the average number of computations required for the color of each pixel, the LSS requires 49.87 logical comparisons and the ALSS requires 26.18.

Even with the tree traversal required by the ALSS algorithm, it still does better in terms of the number of comparisons than the LSS algorithm. In addition, the distance computations, which can be computed as the integer distance squared, are more costly than the logical comparisons. The reduction in distance computations is more than half. And given that the average image size in our experiment was 432942 pixels, the actual running time of the ALSS should be much faster than the LSS algorithm.

I also computed information relavent to one algorithm or the other. For example, I discovered that the average number of lists created by the LSS algorithm was approximately 223 and that the average total path length for the k-d tree in the ALSS algorithm was 9.

4.4.3 Execution Analysis

To verify my results from the previous section, I computed the execution time for each of the 50 images using the ALSS and LSS algorithms. The time was only computed for the actual dithering process and the preprocessing required by the ALSS algorithm.

These experiments were performed on an Intel(tm) Pentium(tm) P5 100Mhz processor based machine running Linux (tm) and X Windows (tm) with 64M of physical memory. The machine was connected to a network but all files (both system and data) were on a local drive. To compensate for the different work loads on the machine, I performed these experiments on three separate occasions and averaged the results. The execution times for the two algorithms are shown in Figure 4.7.

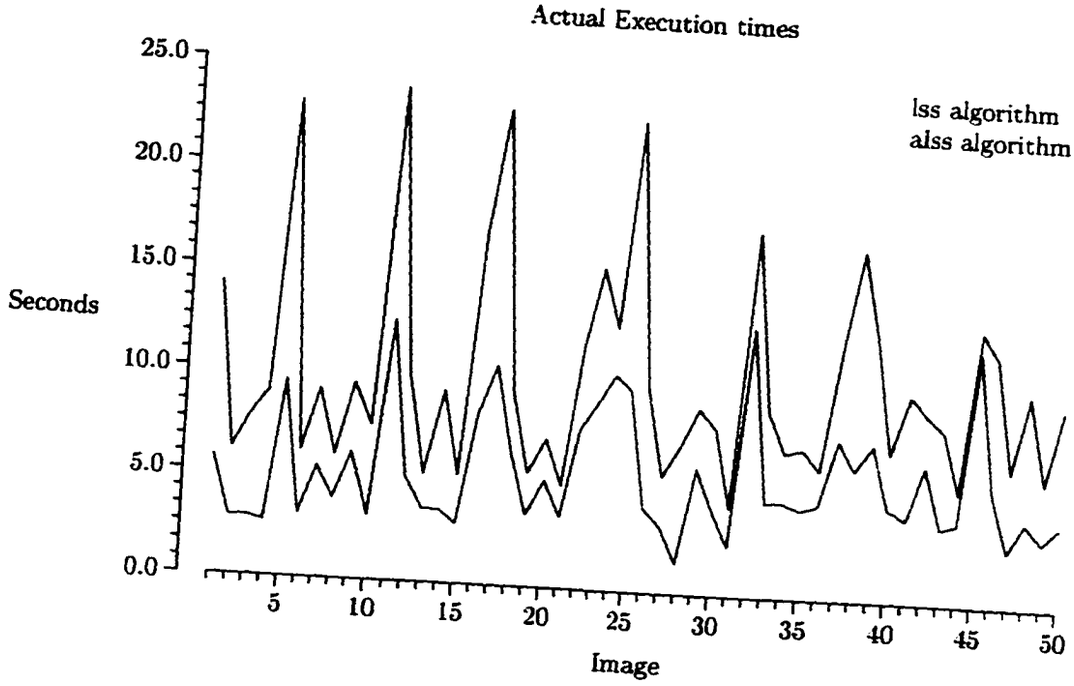


Figure 4.7: Actual running times of the ALSS and LSS algorithms.

Chapter 5

Improvements in the Dithering Process

5.1 Introduction

Many hardcopy devices are limited in the number of colors that they can produce. Consider a bilevel printer which can only produce black and white images. When a grayscale image is to be displayed on such a device, some technique must be used to represent the various shades of gray. The common approach is to construct the image using patterns of black dots to generate the perception of varying shades of gray. This process is known as *halftoning* or *dithering*.

The image produced by the quantization process using a recoloring technique can contain artifacts such as contouring. It has been shown that these artifacts can be reduced using certain dithering techniques to smooth the image by blending neighboring pixels. This has the effect of fooling the eye and brain into perceiving different color tones where none

actually exist. The use of an image smoothing step can be used in place of the simple recoloring step in the quantization process. This technique can greatly improve quantized images for both hardcopy and display devices.

In this chapter, I review some of the popular halftoning and dithering techniques used with both hardcopy and display devices. While the main focus of this chapter is on the use of dithering techniques for display devices, no review is complete without some background on those techniques used with hardcopy devices.

I then present several refinements of my own to the error diffusion dithering technique to reduce the actual running time for use in real-time and interactive systems. I then present my plan for parallelizing the error diffusion algorithm and show that the results for use on a display device are similar to those produced by the original algorithm.

5.2 Previous Work

5.2.1 Halftoning and Dithering Techniques

The original halftoning techniques were applied to grayscale images. With the advent of color printers, the need arose for color techniques. The algorithms used with grayscale images are generally applied to both grayscale and color images. Thus, we introduce the various algorithms in terms of grayscale images. After this introduction, we then focus on their use with color images.

5.2.1.1 Ordered Dithering

To produce different gray levels, a pattern of black dots within a matrix is used to represent a single pixel within the image. By varying the number of black dots within the matrix, the perceived gray-level is varied [6, 20, 34, 98]. For example, consider the 2×2 matrices in Figure 5.1. By replacing each pixel in the image with one of these patters, five different gray-levels can be produced on a bilevel device.

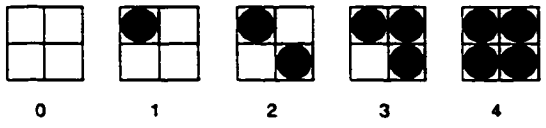


Figure 5.1: Sample 2×2 pattern matrices used with ordered dithering.

To increase the number of producible gray-levels, the size of the matrix is increased. The common size of an ordered dither pattern is 8×8. Large patterns are not usually a problem on hardcopy devices since the number of dots per inch is normally quite high. On a display device, however, large patterns result in sacrificing spatial resolution for color resolution when the size of the image is to remain the same.

Several restrictions on the construction of the dot patterns have been identified [20, 53, 80]. First, the dots should be adjacent, not isolated. This results in better production on some laser printers and printing presses. Second, the pattern of dots should grow outward from the center. This has the affect of creating larger and larger dots for darker intensities. Lastly, the various patterns should produce a growth sequence. That is, if a dot within the matrix is on for an intensity level of j, it should also be on for all levels k, where $k > j$.

Several schemes have been developed which attempt to generate good pattern matrices [36, 39, 69, 88]. Two common 3×3 matrices are illustrated in Figures 5.2(a) and (b). Matrix (a) is due to Foley and van Dam [20] while matrix (b) was developed by Newman and Sproull [68]. The most commonly used 4×4 pattern matrix, illustrated in Figure 5.2(c) is due to Bayer [6]. For any of these matrices, the pattern for a given gray level l, is constructed by setting all the dots within the pattern with a matrix value less than l.

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix} \quad \begin{bmatrix} 7 & 2 & 6 \\ 3 & 0 & 1 \\ 5 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

(a) (b) (c)

Figure 5.2: Pattern matricies can be constructed using different schemes. These matricies illustrate the construction of several of the more popular schemes. For a given gray level, l, all dots within the pattern with a value of less than l are set.

Bayer's solution does not satisfy all of the requirements of a good pattern matrix as described above. However, it was designed so that any sized pattern matrix could be created recursively. Bayer defines the 2×2 pattern matrix to be

$$\mathbf{M}^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

which also defines the patterns in Figure 5.1. To create a larger size pattern matrix, a recurrence relation is used to compute the matrix $\mathbf{M}^{(n)}$ from $\mathbf{M}^{(n/2)}$

$$\mathbf{M}^{(n)} = \begin{bmatrix} 4\mathbf{M}^{(n/2)} + 0\mathbf{U}^{(n/2)} & 4\mathbf{M}^{(n/2)} + 2\mathbf{U}^{(n/2)} \\ 4\mathbf{M}^{(n/2)} + 3\mathbf{U}^{(n/2)} & 4\mathbf{M}^{(n/2)} + 1\mathbf{U}^{(n/2)} \end{bmatrix}$$

where $\mathbf{U}^{(n)}$ is an $n \times n$ matrix of ones

$$\mathbf{U}^{(n)} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \dots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}.$$

The 8×8 pattern matrix computed by applying Bayer's method is illustrated in Figure 5.3.

$$\begin{bmatrix} 0 & 32 & 8 & 40 & 2 & 34 & 10 & 42 \\ 48 & 16 & 56 & 24 & 50 & 18 & 58 & 26 \\ 12 & 44 & 4 & 36 & 14 & 46 & 6 & 38 \\ 60 & 28 & 52 & 20 & 62 & 30 & 54 & 22 \\ 3 & 35 & 11 & 43 & 1 & 33 & 9 & 41 \\ 51 & 19 & 59 & 27 & 49 & 17 & 57 & 25 \\ 15 & 47 & 7 & 39 & 13 & 45 & 5 & 37 \\ 63 & 31 & 55 & 23 & 61 & 29 & 53 & 21 \end{bmatrix}$$

Figure 5.3: Bayer's scheme to create an 8×8 pattern matrix.

5.2.1.2 Error Diffusion

The ordered dithering technique has the disadvantage of introducing unsightly patterns in the resulting image. This usually results from the choice of a poor pattern matrix for the given image. When used with high-resolution printers or printing presses, however, the dot ordered method produces very good results [20, 44]

A second method for creating a halftone image, known as *error diffusion*, was introduced by Floyd and Steinberg [19]. Error diffusion has the advantage of creating halftone images in which the dots appear to be placed at random [96].

The error diffusion method does not use a pattern matrix that is tiled over the image to create various gray levels. Instead, a simple test is used to determine whether each pixel

is turned on (black) or off (white). The result is the appearance of a random placement of dots resulting in a multi gray level image.

The error diffusion method is a three step process which is performed for each pixel of the input image, I, by traversing the image one line at a time from left to right. First, it is determined whether the output pixel, p, should be on (black) or off (white). If the gray level value for the given input pixel, x, is less than half the full range of input gray-level values, then the pixel in is on, otherwise, it is turned off,

$$p = \begin{cases} 0, & \text{if } x < 0.5 \\ 1, & \text{otherwise} \end{cases}.$$

Next, the difference (error) between the original gray-level value and the output value is computed.

$$e = x - p.$$

Finally, the error, e, is spread out in different proportions to nearby pixels that have not yet been processed. A weight matrix is commonly used to specify the exact proportions of the error to be distributed and the neighboring pixels to be affected.

A number of weight matrices have been designed for use with the error diffusion dithering technique. The most common of these is that of Floyd and Steinberg [19], which uses a 3×3 weight matrix that distributes the error to only four neighboring pixels. In Figure 5.4, the current image pixel is indicated by a (•). The pixels to which a fraction of the error is distributed and the proportion of the error distributed is indicated by non-zero weights.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \bullet & \frac{7}{16} \\ \frac{3}{16} & \frac{5}{16} & \frac{1}{16} \end{bmatrix}$$

Figure 5.4: The popular Floyd-Steinberg weight matrix for use with the error diffusion dithering technique.

Some schemes distribute the error difference over 12 neighboring pixels. Two of these well known weight matrices are due to Jarvis, et al [39] and Stucki [93] as illustrated in Figure 5.5.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \bullet & \frac{7}{48} & \frac{5}{48} \\ \frac{3}{48} & \frac{5}{48} & \frac{7}{48} & \frac{5}{48} & \frac{3}{48} \\ \frac{1}{48} & \frac{3}{48} & \frac{5}{48} & \frac{3}{48} & \frac{1}{48} \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \bullet & \frac{8}{42} & \frac{4}{42} \\ \frac{2}{42} & \frac{4}{42} & \frac{8}{42} & \frac{4}{42} & \frac{2}{42} \\ \frac{1}{42} & \frac{2}{42} & \frac{4}{42} & \frac{2}{42} & \frac{1}{42} \end{bmatrix}$$

(a) (b)

Figure 5.5: Two additional error diffusion weight matricies are the Jarvis, Judice, and Ninke matrix (a) and the Stucki matrix (b).

Several problems with the error diffusion methods have been identified [44, 98]. The most serious of which is that "shadows" and "ghosts" tend to be introduced. For example, when the image being processed is that of a person's face, a shadow of their hairline can sometimes be seen in the middle of their forehead or on the side of their face.

Floyd discovered that the shadows tend to disappear in grayscale images if the intensities are rescaled [44]. He suggests replacing each input pixel, x, by the value $0.1 + 0.8 \cdot x$. The affect of this rescaling is simply a contrast stretch of the image. Ulichney [98] found that the

"shadows" can be reduced by changing the image traversal process. Instead of traversing each line of the image from left to right, he suggests that the traversal order alternate between lines. On the first line and each alternate line, the traversal would be from left to right; on each line inbetween those, it would be from right to left. This method of image traversal, termed a *serpentine scan*, allows the error values to be distributed in more directions.

5.2.1.3 Dot Diffusion

The error diffusion method tends to produce better results than the ordered dither method. However, the ordered dither method is completely parallel while error diffusion is serial. In an attempt to combine the best features of these two methods, Knuth [44] developed the *dot diffusion* technique. This technique computes and distributes the error difference like error diffusion but also uses a matrix to limit the area of pixels to which the error can be distributed.

The dot diffusion algorithm subdivides the input image, I, into $n \times n$ sized sub parts or tiles. A technique similar to that in the error diffusion technique is then used on each image tile to set a pattern of pixels. The dot diffusion method uses an $n \times n$ matrix of class numbers similar to that used with the ordered dither technique. Sample 4×4 and 8×8 class matrices are illustrated in Figure 5.6. Here the matrix cells represent a pixel number or class within the tile and not a gray-level value.

Given a class matrix, each tile of the image is processed. For each pixel class within a given tile, $\{0, 1, \ldots, n^2 - 1\}$, the corresponding output image pixel, p, is turned on (black) if the value of the corresponding input image pixel, x is less than half the maximum gray

$$\begin{bmatrix} 14 & 13 & 1 & 2 \\ 4 & 6 & 11 & 9 \\ 0 & 3 & 15 & 12 \\ 10 & 8 & 5 & 7 \end{bmatrix} \qquad \begin{bmatrix} 34 & 48 & 40 & 32 & 29 & 15 & 23 & 31 \\ 42 & 58 & 56 & 53 & 21 & 5 & 7 & 10 \\ 50 & 62 & 61 & 45 & 13 & 1 & 2 & 18 \\ 38 & 46 & 54 & 37 & 25 & 17 & 9 & 26 \\ 28 & 14 & 22 & 30 & 35 & 49 & 41 & 33 \\ 20 & 4 & 6 & 11 & 43 & 59 & 57 & 52 \\ 12 & 0 & 3 & 19 & 51 & 63 & 60 & 44 \\ 24 & 16 & 8 & 27 & 39 & 47 & 55 & 36 \end{bmatrix}$$

Figure 5.6: Two of Knuth's class matricies for use with his dot diffusion technique.

level value; otherwise, p is turned off (white). Next, like the error diffusion technique, the error difference is computed which is simply the difference between the original and output pixel values, $e = x - p$. Finally, the error is distributed to the neighbors of the current pixel within the given tile, whose class numbers have not yet been processed.

In distributing the error difference to the neighbors of a given pixel, a weight matrix is used as in the error diffusion method. In the dot diffusion method, the error difference could be distributed to all the neighbors of a pixel, to only a subset of the neighbors, or to a single neighbor. The weight matrix used by Knuth is illustrated in Figure 5.7. The total value distributed can not exceed the error value. Thus, the value of w has to be adjusted such that the sum of the distributed weight equals 1.

$$\begin{bmatrix} \frac{1}{w} & \frac{2}{w} & \frac{1}{w} \\ \frac{2}{w} & \bullet & \frac{2}{w} \\ \frac{1}{w} & \frac{2}{w} & \frac{1}{w} \end{bmatrix}$$

Figure 5.7: The weight matrix used with Knuth's dot diffusion algorithm. The error difference may not be distributed to all the neighbors. Thus, the value of w must be adjusted so that the total weight equals 1.

The dot diffusion technique has the advantage that it can be implemented as a parallel process, unlike the error diffusion method. Studies have shown that the dot diffusion method produces better images, in appearance, than the other two methods [44].

5.2.2 Dithering Color Images

The dithering process is also needed for the printing of color images. Most hardcopy printing devices use the CMY [20] color space and subtractive color mixing. In this process, the three colors cyan, magenta, and yellow are overlaid or mixed to produce different colors.

The ordered dithering process is the most common approach used in printing color images [20, 33, 69]. Instead of using a single pattern, however, there is a different set for each of the primaries. Using different pattern sets increases the chances that patterns will not form from the dot placement. In addition, the multiple pattern sets help to decrease the amount of ink sprayed onto a particular spot. There are many ways to create the additional pattern sets. Figure 5.8 illustrates four patterns for a given brightness level. Given the first pattern, each successive pattern was obtained by rotating the previous pattern by 90 degrees.

Figure 5.8: Four patterns created by rotating the original 90 degrees.

A color image is drawn by dealing with the three component values separately [33]. For a given pixel, a pattern is chosen from each pattern set based on the range in which the component value lies. Selected patterns are then laid or painted on top of one another

to create the color for the given pixel. Using a 2×2 pattern, 125 different colors can be produced [33].

The error diffusion method has also been used to produce halftoned color images [26]. Each color component is processed separately to decide whether a given pixel should be painted with the corresponding color. The drawback to this method for use with color hardcopy devices is that large amounts of ink may result in certain areas of the image.

My literature search for this dissertation did not reveal the use of the dot diffusion method with color images. But it would appear that it too could be used in a manner similar to the ordered dither and error diffusion methods.

5.2.3 Image Smoothing by Dithering

The image produced by the quantization process, as described in the previous chapters, can contain artifacts resulting from the use of fewer colors. The most common artifact is that of contouring. Contouring occurs when the color intensities in two adjacent regions differ significantly without a gradual change from one intensity to the next.

Consider an example. The image in Figure 2.2(a) is the original image containing 53964 colors, while the image in Figure 2.2(b) is the recolored version reduced to 256 colors using the median cut algorithm. Notice the contouring artifacts on the spheres of the object in the recolored quantized version. In the original image, this part consisted of numerous color tones which created a gradual change from the outside of the sphere to the inside. With only a limited number of colors available for the recoloring, the visual effect could not be recreated.

5.2.3.1 Pixel Blending

Artifacts in an image degrade the quality of that image. One approach to reducing these artifacts is to smooth the image by blending neighboring pixels. A common approach is to use a modified version of the error diffusion dithering technique [26, 35, 96]. Here, the purpose of the dithering technique is not to create patterns of dots for presentation on a hardcopy device, but instead to blend the pixels to create the illusion of color tones where none actually exist. This technique is used for both hardcopy and display devices without the loss of pixel resolution.

Most authors recommend the use of the error-diffusion dithering technique due to Floyd and Steinberg [19, 33, 96]. While the Floyd-Steinberg algorithm was originally designed to work with grayscale images printed on a bilevel device, it works well, applied component by component, with color images for display on a colored display device.

Given the quantizer $Q_{P,Y}$, the error-diffusion dithering technique can be used to produce more visually pleasing images than the simple recoloring technique introduced in Chapter 1 [35, 96]. The blending dither technique is a three step process. First, at each pixel, the input color is mapped to a representative color based upon the partition P. The current output pixel is then set to the representative color. Next, the error difference is computed for each color component between the input color and output color. Finally, the error difference is distributed to the neighboring pixels that have not yet been processed. This last process is the same as for grayscale images presented earlier. The only difference, is that the error value has to be distributed for each color component.

To illiminate costly floating-point arithmetic, the error diffusion technique can be implemented using integer arithmetic without loss or degradation of image quality. One such implementation is provided in Appendix A.

5.2.3.2 Pixel Scanning

The error-diffusion dithering technique processes the pixels one at a time by scanning each row from left to right. This scanning approach is known as a *raster scan* which is named after the method used by modern CRT devices for setting pixels on the screen.

Ulichney has shown that better results can be achieved if a *serpentine scan* is used instead [98]. With a serpentine scan, as illustrated in Figure 5.9, the pixels on even numbered rows are scanned left to right, while those on odd numbered rows are scanned right to left.

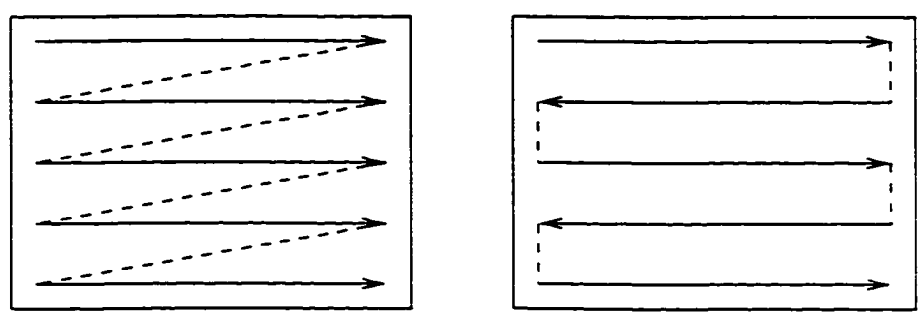


Figure 5.9: A raster scan versus a serpentine scan. A serpentine scan can produce better results since the error is distributed in a more evenly fashion.

5.2.3.3 Error Clamping

A second modification that can improve the results of the error-diffusion dithering technique is to limit or clamp the total error value that is distributed to a single pixel. Limiting the

error value prevents it from getting too large and adversely affecting far away parts of the image [33, 9].

This technique is used with the JPEG image storage format. Instead of simply distributing the error value to the neighboring pixels, the value is stored until the neighboring pixel is processed. Before a representative color is found for a given pixel, the total error value that is supposed to be added to this pixel is clamped using some clamping method. After the error is clamped, it is then added to the input color, component by component. A representative color from Y is then found for the error adjusted color [9].

There are four basic approaches to clamping the error value. The original method is to clamp the error at 255 or -255 as shown in the following function

$$\text{clamp}_1(l = 255, x) = \begin{cases} x, & x \geq 0 \\ l, & x > l \\ -\text{clamp}_1(l, |x|), & x < 0 \end{cases}$$

Two other approaches clamp the error value to 16 (or -16) and 32 (or -32), respectively [33]. Thus, we would use $\text{clamp}_1(32, x)$ and $\text{clamp}_1(16, x)$ respectively. Aaron Giles has recommended the use of a pseudo stepping function to clamp the total error value [9]

$$\text{clamp}_2(x) = \begin{cases} x, & 0 \leq x \leq 16 \\ (x-1)/2 + 8, & 16 < x \leq 48 \\ 32, & 48 < x < 256 \\ -\text{clamp}_2(|x|), & -256 < x < 0 \end{cases}$$

There are no known statistics or experiments comparing these four methods. Simple experiments, however, do show that Giles' function produces better results than the others. Giles' error clamping function has become the defacto standard and will be used throughout the remainder of this paper.

The image in Figure 2.2(c) is the dithered version of that in Figure 2.2(a). The Floyd-Steinberg weight matrix was used to distribute the weights as was a serpentine scan. The error values were clamped using the clamping function suggested by Giles.

5.3 Analysis of Quantization with Dithering

In Chapter 3, I presented results of the popular quantization algorithms using the simple recoloring scheme. Better results can be achieved, however, using the error diffusion dithering technique instead of recoloring. In this section, I present the results of several experiments involving some of the quantization algorithms presented earlier.

5.3.1 Dithering with Original Algorithms

Heckbert showed that the use of error diffusion dithering could improve the results of his median cut algorithm. His implementation used the color reduction scheme for both recoloring and dithering. The center cut algorithm does produce better results when dithering instead of recoloring, but the results are still not as good as the other algorithms. My color cut algorithm can also be used with dithering since a complete partition is created for P.

The original implementation of the octree quantization algorithm can not be used with dithering since the recoloring is performed by searching for a given color in the octree. When recoloring, each search is guaranteed to end at a leaf node. For dithering, colors other than those appearing in the image will occur and it is not guaranteed that a search for such a color will end at a leaf node.

I conducted several experiments involving the use of dithering with the several algorithms. First, I compared the results between recoloring and dithering for the median cut

and center cut algorithms where $k = 256$. The results for this experiment are illustrated in Figure 5.10. As can be seen from the graph, dithering greatly improves the results for both algorithms. Though, the results for the center cut algorithm are still not as good as those for the median cut algorithm.

5.3.2 The Color Cut Algorithm

I also tested my color cut algorithm with dithering and compared it to the results of the algorithm with simple recoloring. The results are illustrated in Figure 5.11. Then I made a comparison between the median cut, center cut and color cut algorithms when used with dithering. The graph in Figure 5.12, which combines some of the plots from Figures 5.10 and 5.11, illustrates this comparison. As in the previous experiment, dithering improves the results over simple recoloring. In addition, dithering with the color cut algorithm produced much better results than the median cut and center cut with dithering.

5.3.3 Modified Octree Quantization Algorithm

The octree quantization algorithm can be modified to work with error diffusion dithering. First, the octree is constructed as in the original algorithm. After processing the entire image, the average color of each leaf node is computed and becomes an entry in Y. For dithering, the octree can no longer be used and thus, it can be destroyed. Instead, my ALSS algorithm can be used to perform the necessary nearest neighbor searches.

The graph in Figure 5.13 illustrates the results of using the modified octree algorithm on the set of test images. A comparison is made between the original octree quantization algorithm and the modified version. I also compare the modified version (with dithering)

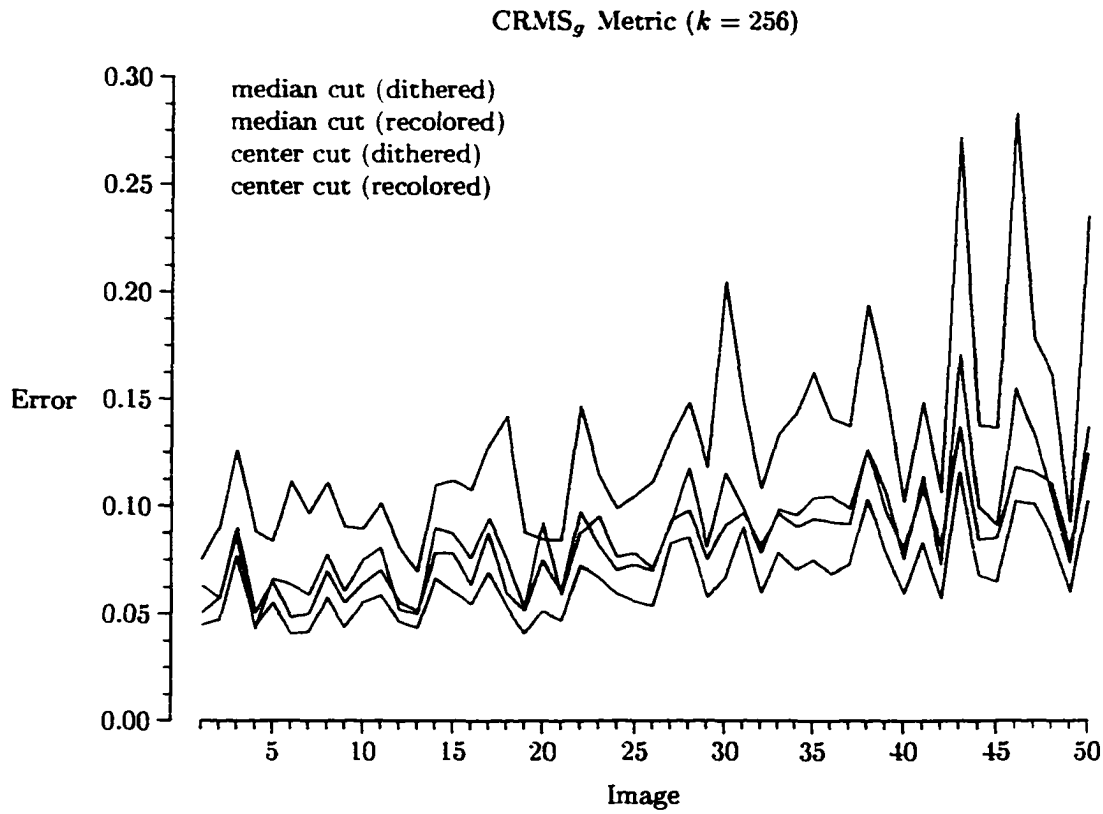


Figure 5.10: A comparison of recoloring versus dithering with the median cut and center cut quantization algorithms.

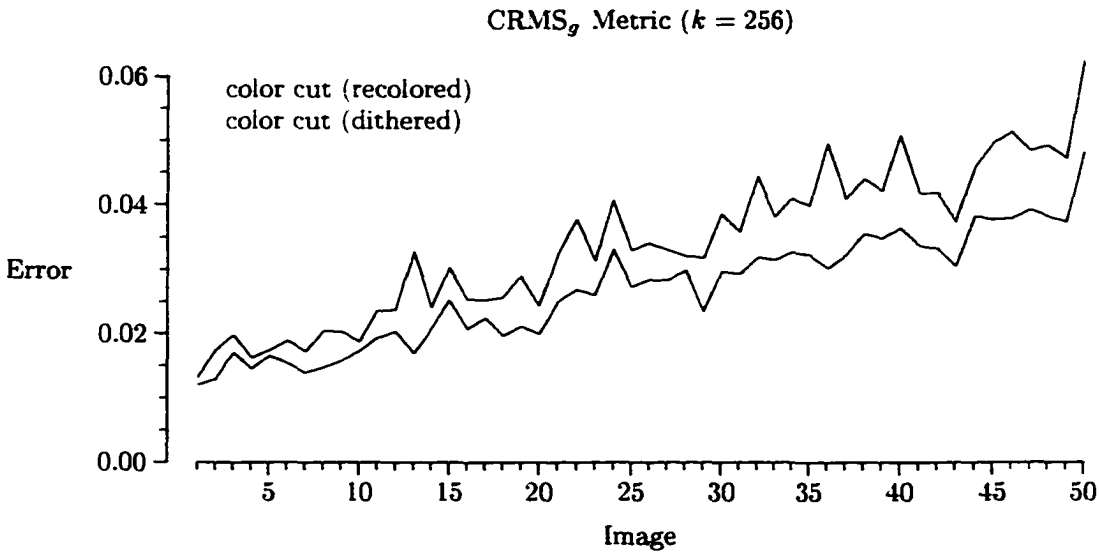


Figure 5.11: A comparison between recoloring and dithering using the color cut algorithm on the set of test images.

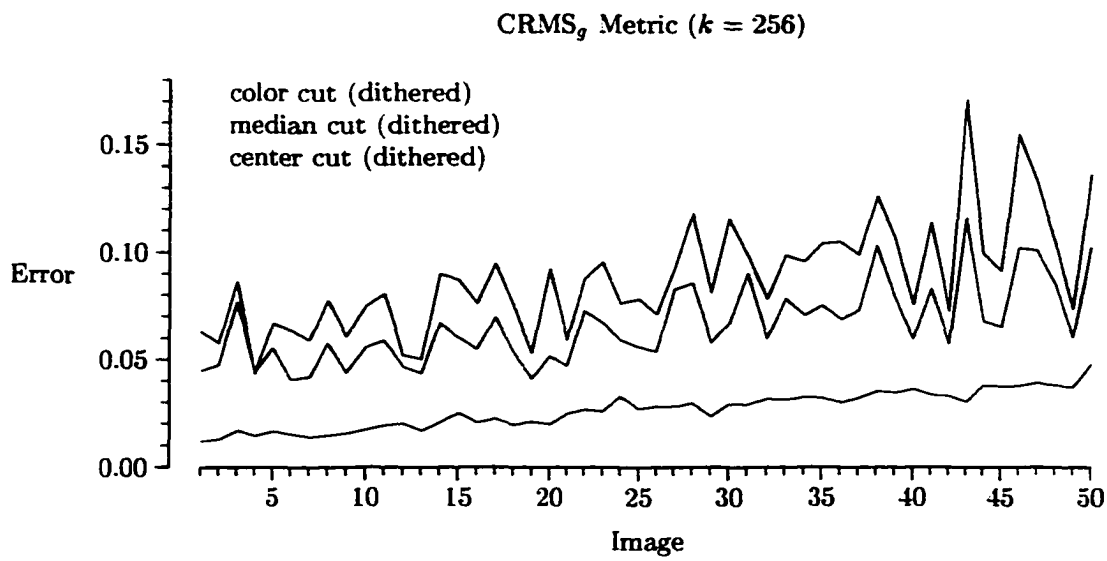


Figure 5.12: A comparison between the dithering results of the three algorithms. These are the same plots used in Figures 5.10 and 5.11.

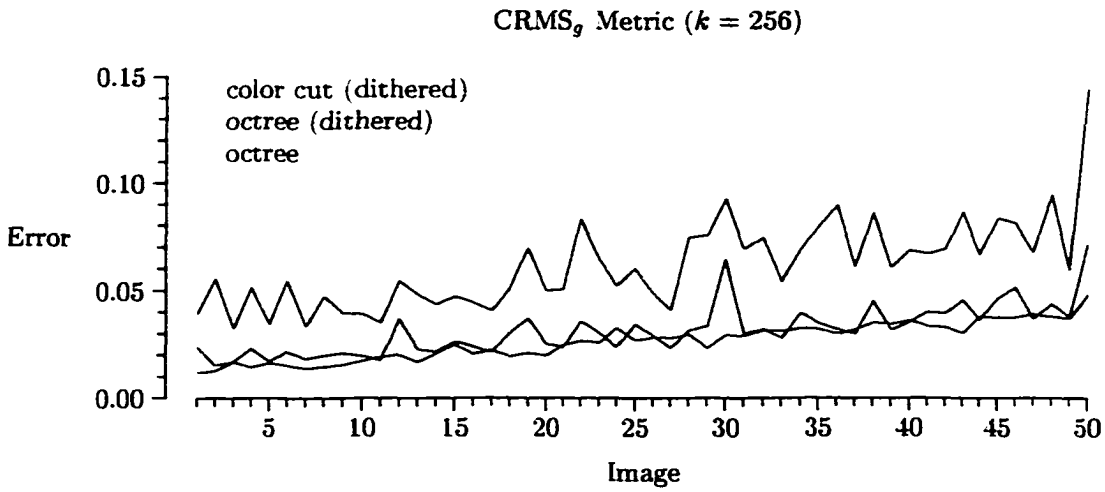


Figure 5.13: A comparison of recoloring versus dithering between the original octree algorithm, the modified octree algorithm and the color cut algorithm.

against the results of the color cut algorithm (with dithering). As can be seen in the figure, the modified octree used with dithering produces results almost as good as those of the color cut algorithm (with dithering).

In some instances, the results from dithering with the modified octree algorithm are better than those from the color cut algorithm. The color cut algorithm, however, is more consistent in producing better images.

5.3.4 Color Cut with Color Reduction

In this section, I present the results of an experiment conducted to evaluate the affects of dithering using the color cut algorithm with a color reduction scheme. I used the same reduction schemes as those in the experiment in Section 3.3.4. In Figure 5.14, the error difference values are plotted for the the three reduction schemes and the original algorithm.

As with recoloring, there is little difference between the one-bit reduction and the original algorithm with no color reduction. This analytical measurement also replicates the results of visual inspection. In fact, for most of the images it is very difficult if not impossible to distinguish visually between the results of the one-bit reduction and those with no reduction.

The results of the larger bit reductions do show a greater difference similar to that when recoloring. However, the results of dithering using the color cut algorithm with the larger bit reductions are still better than recoloring with no bit reduction. This is illustrated in Figure 5.15. Thus, good results can be achieved using a 2-2-2 bit reduction with dithering to reduce the execution time.

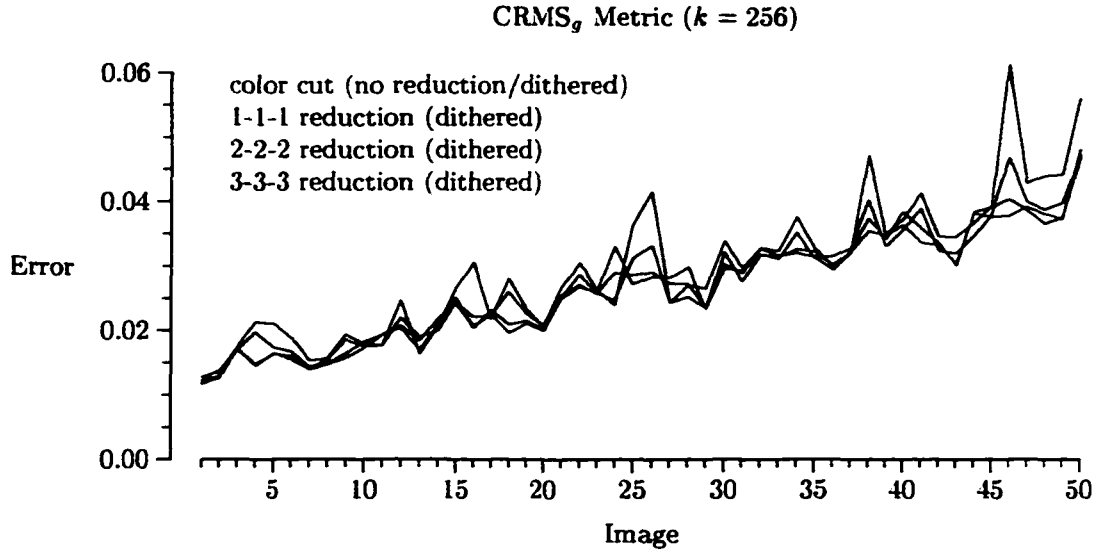


Figure 5.14: The results of dithering using the color cut algorithm with color reduction. The results from the color reductions are compared to the results of using dithering with the color cut algorithm and no color reduction.

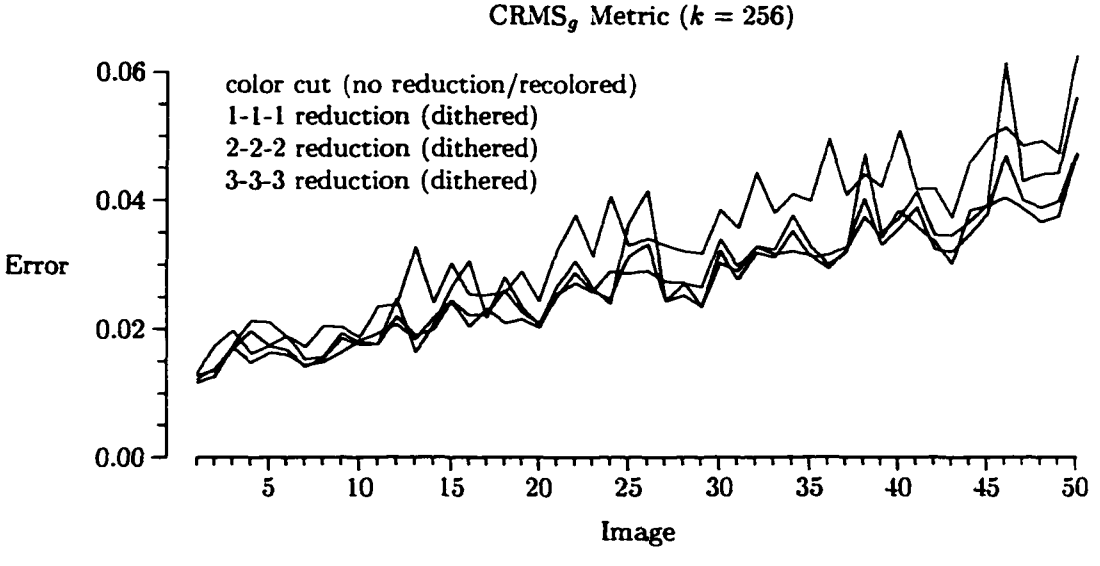


Figure 5.15: The results of dithering using the color cut algorithm with color reduction is compared against the algorithm with no reduction and simple recoloring.

5.4 Parallelizing the Error Diffusion Algorithm

One benefit of the dot diffusion and ordered dithering techniques over error diffusion is that they are parallel while error diffusion is serial. Error diffusion, however, is the most commonly used technique for dithering images on a display device. And it is always implemented as a serial process. But does this mean that the error diffusion algorithm can not be implemented as a parallel process? At first it would appear so.

Consider the scanning and error distribution. The elements of the image are scanned from the top left to the bottom, one row at a time. The error is distributed to those neighbors of an element that have not yet been processed. Thus, before a given element can be processed, all of the elements preceding the given element (in a serial fashion) must be processed. This would be nearly impossible to implement in a parallel fashion.

But what if the error diffusion algorithm could be implemented in a fashion similar to the dot diffusion and ordered dithering techniques? These techniques process small blocks of the image independent of the other blocks. Thus, the various blocks can be processed at the same time by different processors.

5.4.1 A Tiling Approach to Error Diffusion

It would appear that any attempt to parallelize the error diffusion algorithm would require the processing of small independent blocks. If the blocking process was used, what type of visual results could be expected? And second, into what size of blocks should the image be sub divided?

I experimented with this concept and found that the error diffusion algorithm could be implemented to process small independent image blocks. The visual results were surprisingly good. In addition, I experimented with various block sizes and found that sizes as small as 8×8 can produce very good results.

To test the blocking approach to error diffusion, I sub divided the image into small blocks of pixels of equal size. Blocks along the right side and bottom were smaller if a full block could not be created due to the size of the image. The color cut algorithm ($k = 256$) with the serpentine scan (using the $\text{clamp}_2(x)$ function) was then applied to each block. Thus, the error was distributed only within each independent image block. The analytical results of my experiments, using different size of blocks, are shown in Figures 5.16 – 5.20.

There was only a noticeable deviation in the analytical difference from that of the original error diffusion algorithm where the block size was 8×8. Though, a visual inspection for this case does not appear to show any difference.

Next, I performed similar experiments where the representative set contained 64 colors. The analytical results from this set of experiments are illustrated in Figures 5.21 – 5.23.

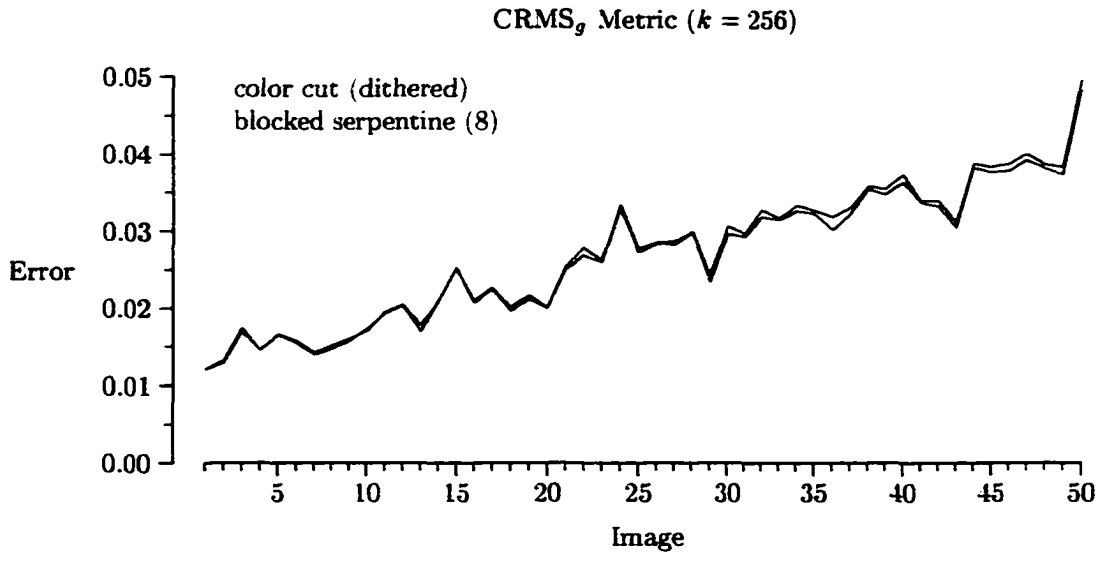


Figure 5.16: Dithering using small image blocks and the Floyd-Steinberg weight matrix. The image was divided into 8 × 8 blocks and then each block was dithered independently.

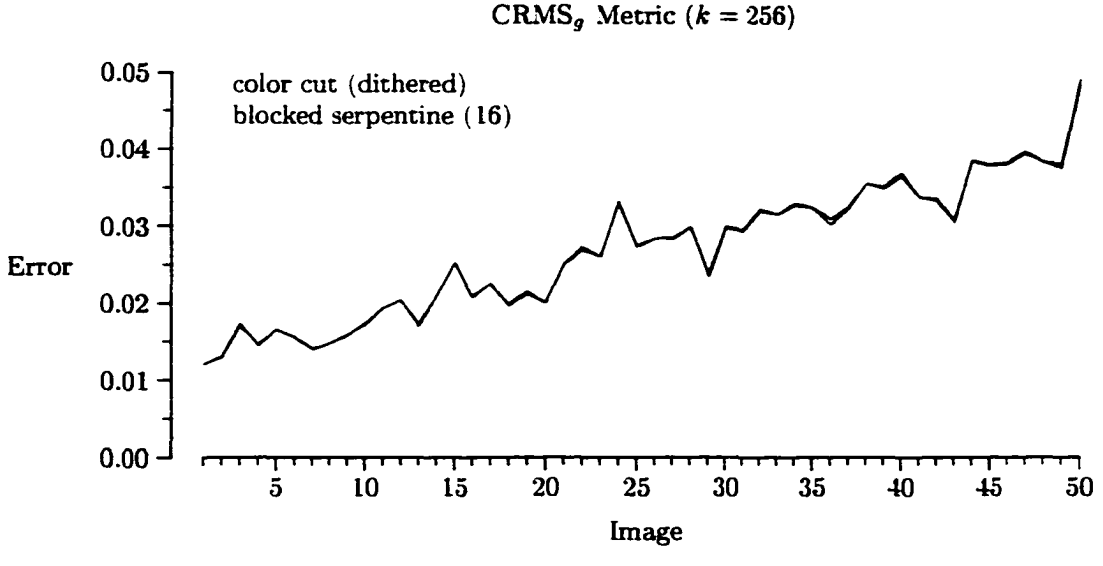


Figure 5.17: Comparison of dithering using small image blocks (size = 16) against dithering of the entire image.

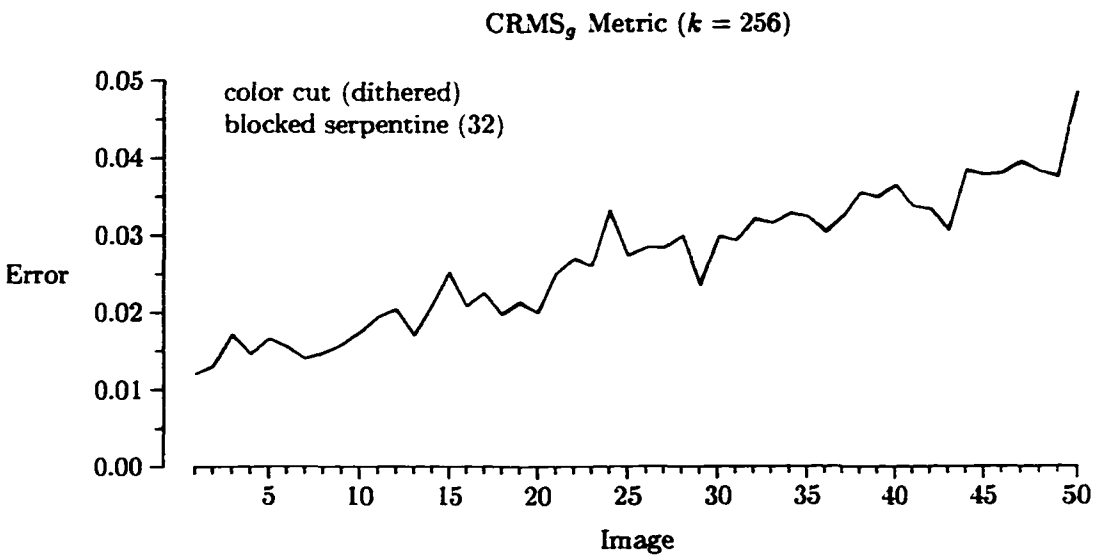


Figure 5.18: Comparison of dithering using small image blocks (size = 32) against dithering of the entire image.

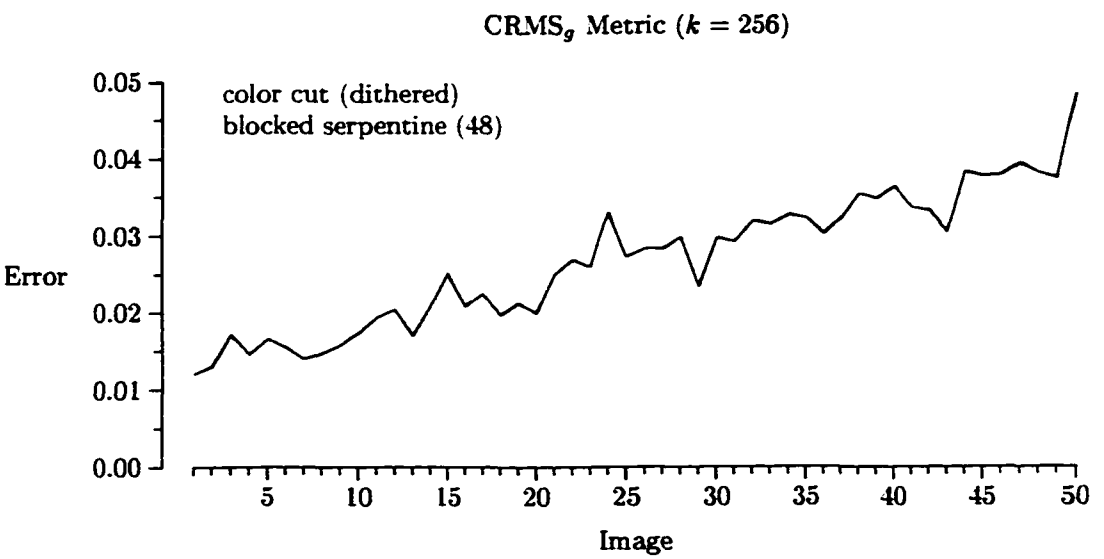


Figure 5.19: Comparison of dithering using small image blocks (size = 48) against dithering of the entire image.

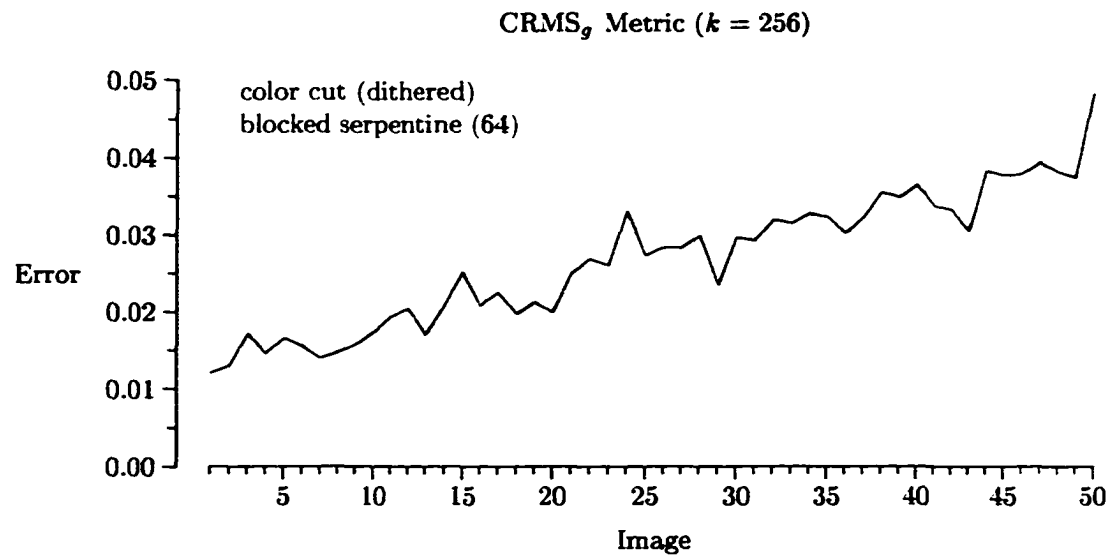


Figure 5.20: Comparison of dithering using small image blocks (size = 64) against dithering of the entire image.

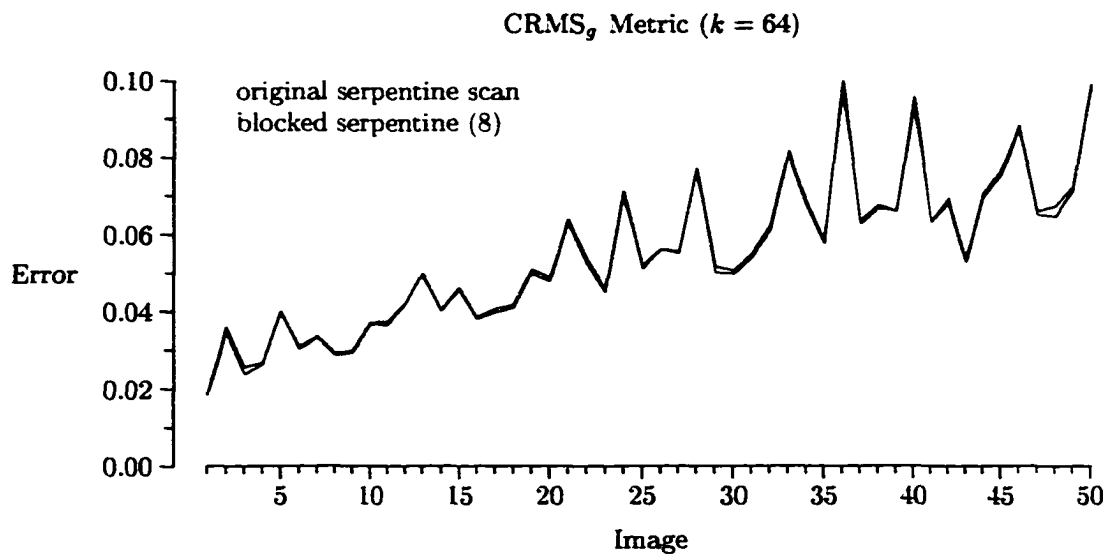


Figure 5.21: Dithering using small image blocks (size = 8, k = 64).

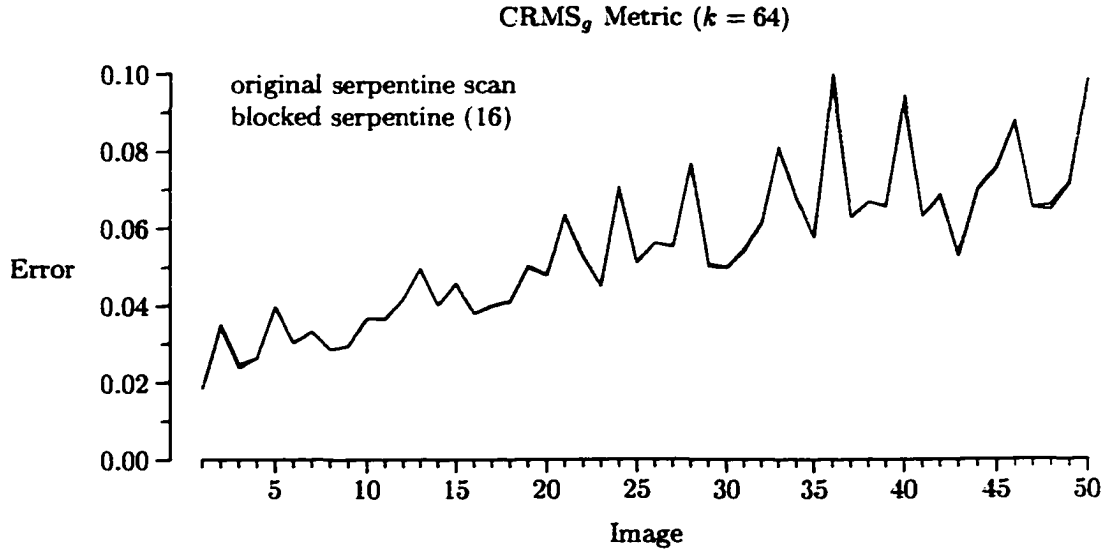


Figure 5.22: Dithering using small image blocks (size = 16, k = 64).

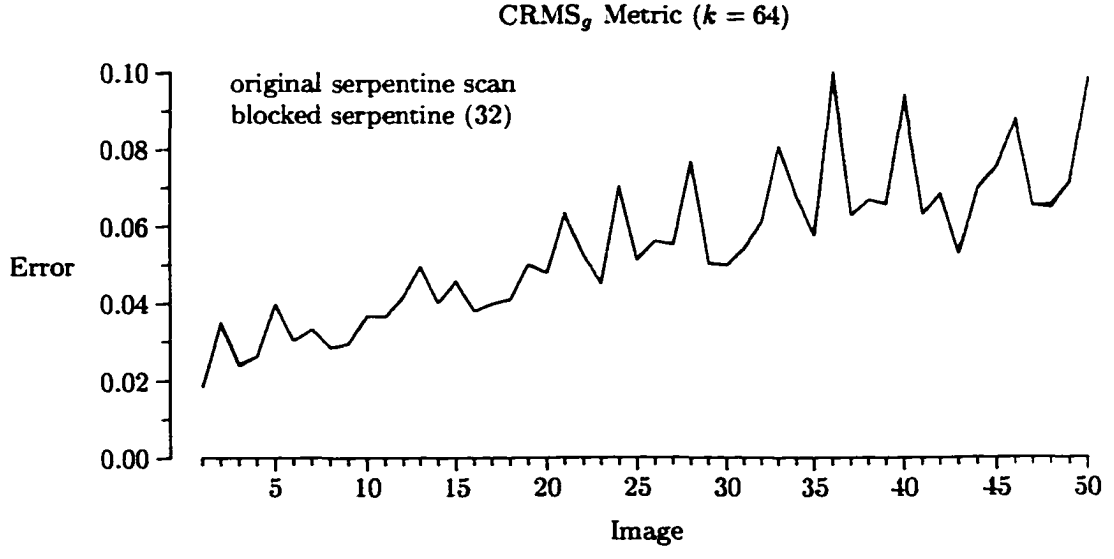


Figure 5.23: Dithering using small image blocks (size = 32, k = 64).

5.4.2 A Parallel Implementation

Based on the analytical and visual results, the error diffusion algorithm can be implemented to process an image using a blocking approach similar to that used by dot diffusion and ordered dithering. Though I do not provide an implementation, the error diffusion algorithm could be implemented as a parallel algorithm and produce similar if not the same results as the traditional serial approach. One could also modify the serial implementation to use the blocking approach to reduce the amount of memory required to process the image. Since this approach would be the same as laying tiles on a floor, I term it *error diffusion by tiling.*

The results in this section, as in the entire chapter, are only valid for display devices. I have not experimented with the use of the blocking approach on hardcopy devices. Though, this does provide a direction for future study.

5.5 A New Weight Matrix

The use of error-diffusion dithering with interactive or real-time applications (i.e., televideo conference, interactive web telecast, and etc.) is commonly avoided due to the computation time required. This is especially true when large images have to be processed. In these cases, time is of the essence.

The error-diffusion technique using the Floyd-Steinberg weight matrix has become the most popular method of dithering color images. I have found, however, that two new weight matrices can produce almost identical results, both visually and analytically, while reducing the required computations.

Consider the classic implementation of the error-diffusion algorithm as described in Appendix B. This implementation, as would any implementation, requires a number of mathematical operations to be performed for each pixel in the application of the weight matrix and error distribution. Specifically, the implementation in Appendix B requires 9 multiplications, 6 additions, and 12 assignments per pixel. While all of these operations are performed on integers, they do add up when processing large images. If the number of operations could be reduced or at least the multiplications reduced to simple bitwise shift operations, the actual running time could be improved.

In order to reduce the number of operations required, I experimented with two new weight matrices for use with error-diffusion. These matrices are illustrated in Figure 5.24. For matrix (a), the mathematical operations can be reduced to 3 multiplications, 3 additions, and 9 assignments; for matrix (b), they are reduced to only 3 assignments. Therefore, if matrix (b) were used in an application, the actual execution time could be reduced. That is of course if it produced acceptable results. The implementations of the error-diffusion algorithm using matrix (a) is illustrated in Appendix B.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \bullet & \frac{2}{4} \\ \frac{1}{4} & \frac{1}{4} & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 \\ 0 & \bullet & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix}$$

(a) (b)

Figure 5.24: Two proposed error-diffusion weight matricies. These matricies reduce the number of computations by reducing the number of distributions and by using values that are powers of two.

5.5.1 Algorithm Comparisons

To determine the type of results obtained using these two weight matrices, I performed several experiments on the same set of test images used throughout this thesis. First, quantizers were designed using the color cut quantization algorithm for both $k = 256$ and $k = 64$. These quantizers were then used to dither the images using the error-diffusion technique. Three sets of dithered images were produced. One set used the Floyd-Steinberg matrix, the other two used my new matrices. The error difference was then computed for each dithered image using the CRMS_g error metric.

These results are plotted in several graphs. First, the graph in Figure 5.25 compares the results ($k = 256$) between the Floyd-Steinberg matrix and my matrix (Figure 5.24(a)) with three weights. While the graph in Figure 5.26 compares the results between the Floyd-Steinberg matrix and my new matrix with two weights (Figure 5.24(b)). Two additional graphs are provided to illustrate (Figures 5.27 and 5.28) the same comparisons but this time with $k = 64$.

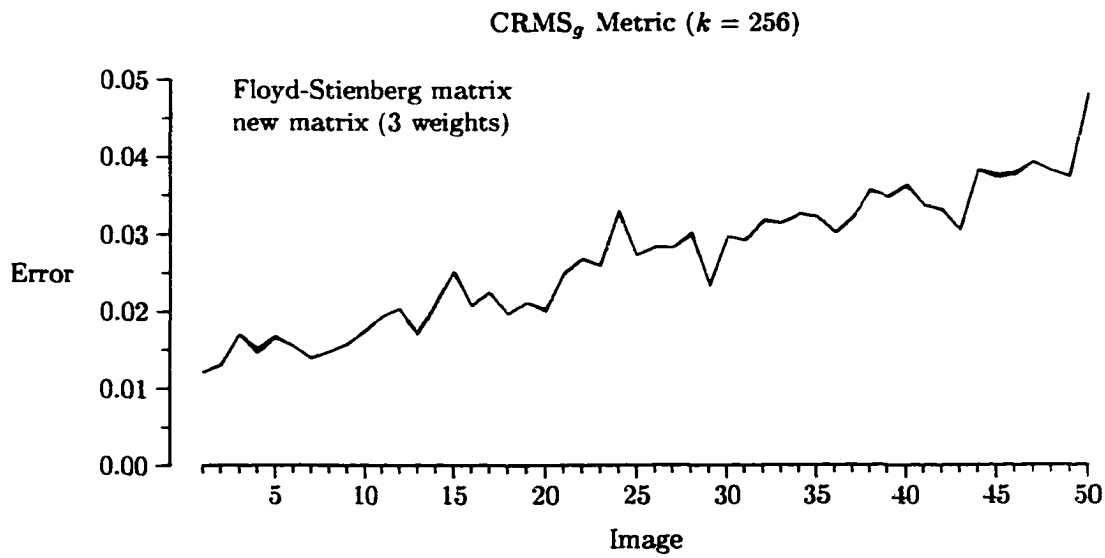


Figure 5.25: A comparison of the Floyd-Steinberg matrix against the three-weight matrix in Figure 5.24(a). The color cut quantization algorithm was used to generate the quantizers for $k = 256$.

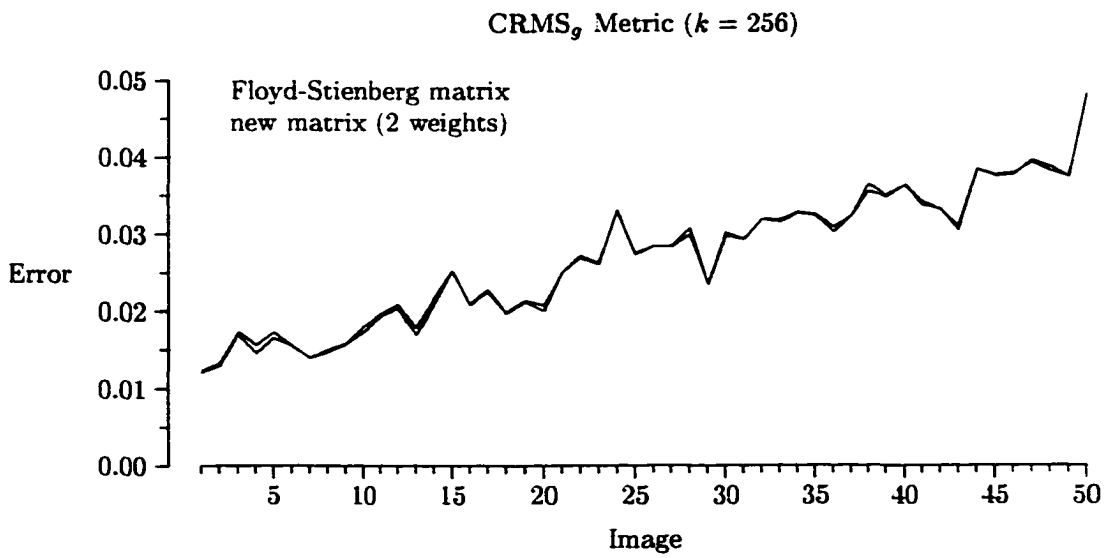


Figure 5.26: A comparison of the Floyd-Steinberg matrix against the two-weight matrix in Figure 5.24(b) for $k = 256$.

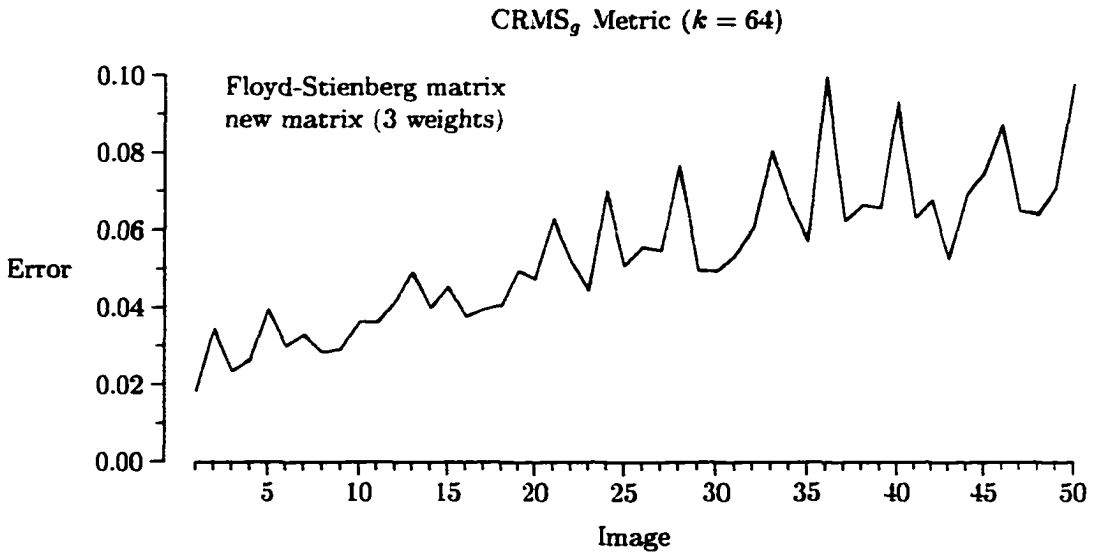


Figure 5.27: The Floyd-Steinberg matrix and the three-weight matrix ($k = 64$).

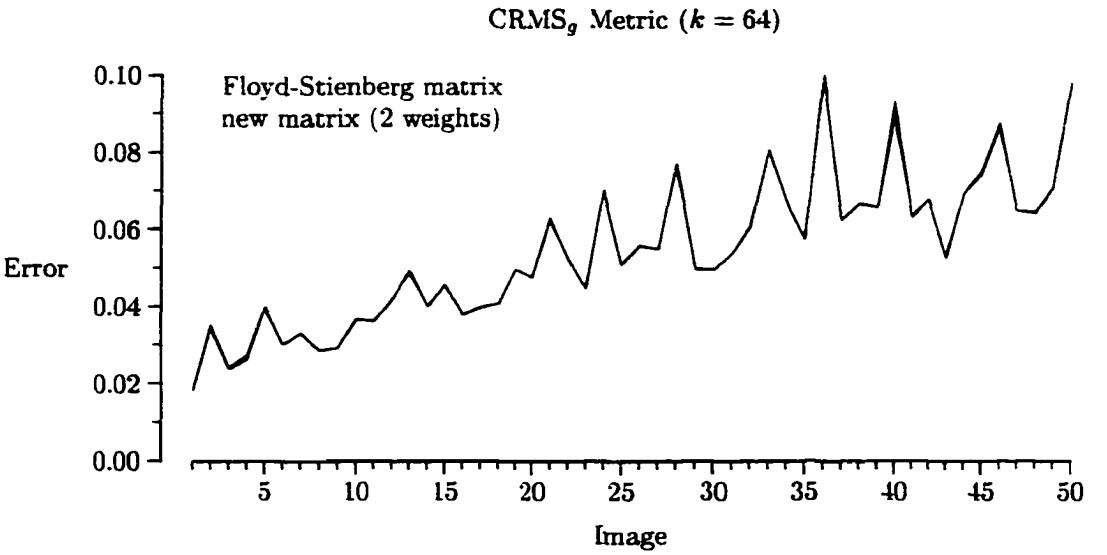


Figure 5.28: The Floyd-Steinberg matrix and the two-weight matrix ($k = 64$).

From these experiments and a visual inspection, there is very little difference between the images resulting from the use of either new weight matrix. Therefore, it would appear that the use of my new matrix with only two weights could be used to improve the execution time without a loss of visual quality. It should be pointed out, however, that the visual inspection was done on a display device.

5.5.2 Execution Analysis

The analytical analysis performed in the previous section shows that the weight matrix used with error diffusion can be reduced to a two-weight matrix and still produce similar visual results. To verify that an improved execution time is achieved by using the two-weight matrix, I computed the execution time for each of the 50 images using my two-weight matrix and the Floyd-Steinberg matrix.

The experiments [1] involved the creation of a color map using the color cut algorithm and the use of the ALSS algorithm for the nearest-neighbor searches. The dithering process was performed using the two functions provided in Appendix B. The time was only computed for the actual dithering process as shown in the code of the two functions. The execution times for the two different weight matrices are shown in Figure 5.29.

5.5.3 The Matrices used with Color Reduction

If the color cut algorithm with color reduction were used with my two-weight matrix, then the execution time could be improved in addition to a reduction in the amount of memory

[1] The experiments were performed on an Intel (tm) Pentium (tm) P5 100Mhz processor based machine running Linux (tm) and X Windows (tm) with 64M of physical memory. The machine was connected to a network but all files (both system and data) were on a local drive.

required. To determine if the results are acceptable, I conducted an experiment using this combination. The error difference values for the same bit-reduction schemes used earlier are plotted in Figure 5.30. Those values are compared against the plot resulting from the use of the original color reduction with dithering using the Floyd-Steinberg matrix.

5.5.4 The Matrices used with Tiling

The new two-weight matrix appears to work well with the normal serpentine scan. To determine the effectiveness of the two-weight matrix with the tiled scan introduced in Section 5.4.1, I performed several experiments. I used the color cut algorithm to produce the quantizer for each of the images in the test set. Next, I dithered the images using the normal serpentine scan and computed the quantization error for each. Then, I dithered the images using the tiled scan with the two-weight matrix. Again, I computed the quantization error for each image. This was done for three tile or block sizes: 8, 16, and 32. The results of these experiments are illustrated in Figures 5.31 through 5.33.

These results are very similar to those when comparing the tiled approach using the Floyd-Steinberg matrix to the normal serpentine approach. Thus, it appears that the tiled approach and the use of the two-weight matrix produces results very similar to those of the original error diffusion algorithm.

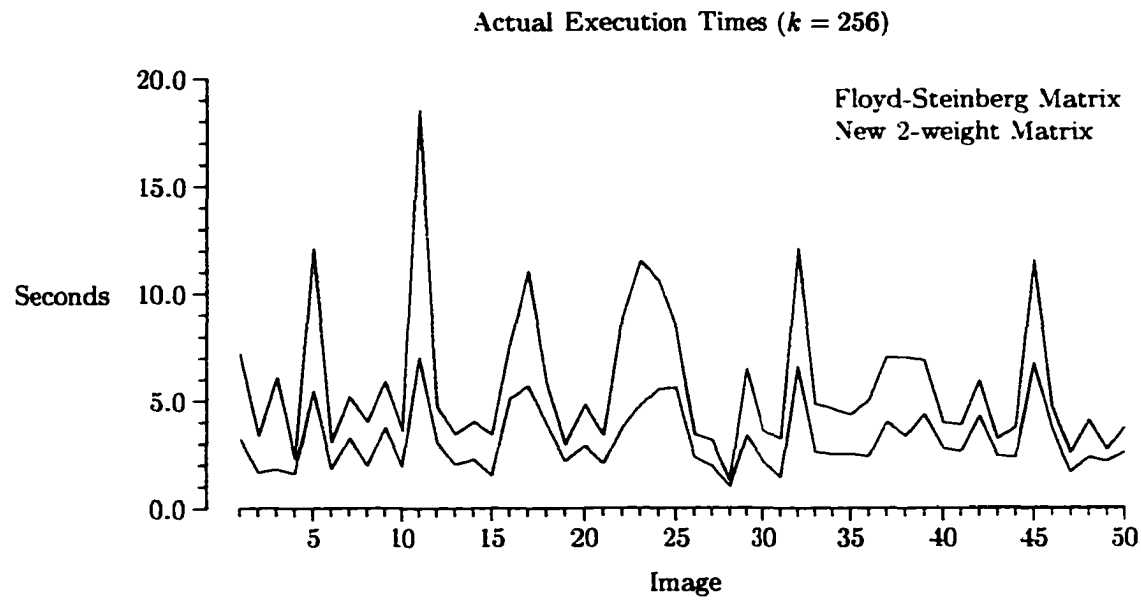


Figure 5.29: Comparison of the actual running times between the use of my two-weight matrix and the original Floyd-Steinberg matrix.

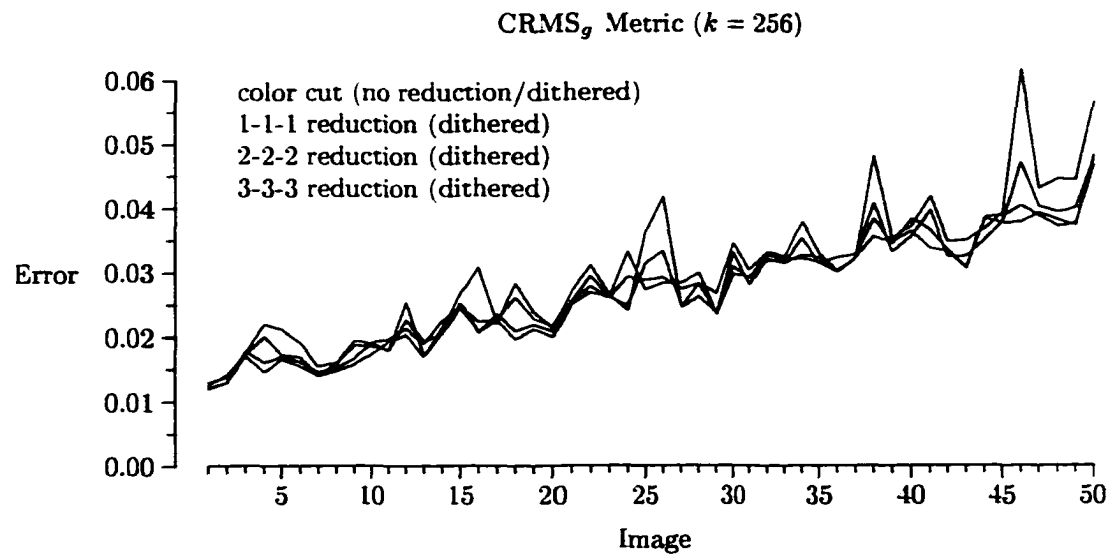


Figure 5.30: Comparing the color cut algorithm with no color reduction (using the Floyd-Steinberg matrix) to the color cut with color reduction using the new two-weight matrix. A serpentine scan was used to dither all images.

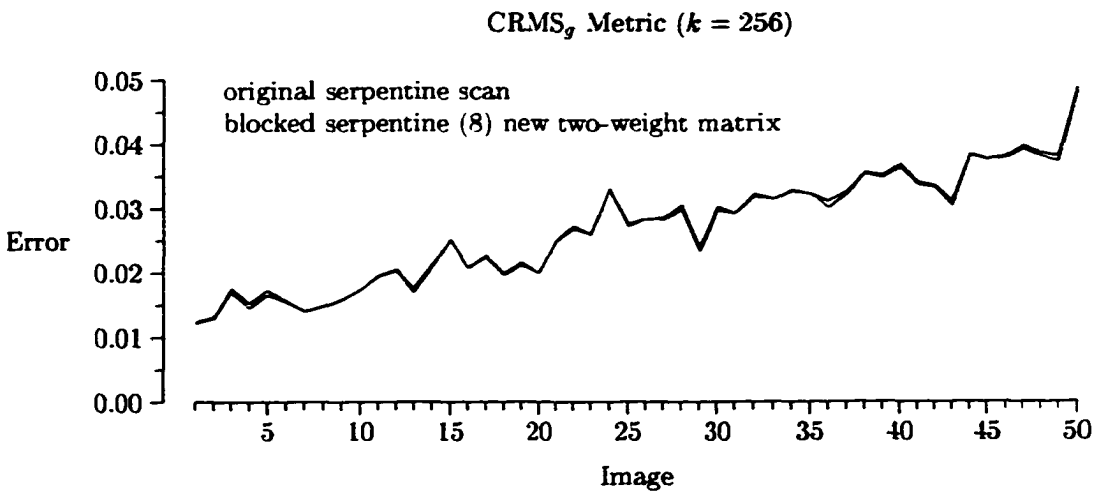


Figure 5.31: Comparing the original serpentine scan using the Floyd-Steinberg matrix with the tiled scan (block size = 8) using the new two-weight matrix.

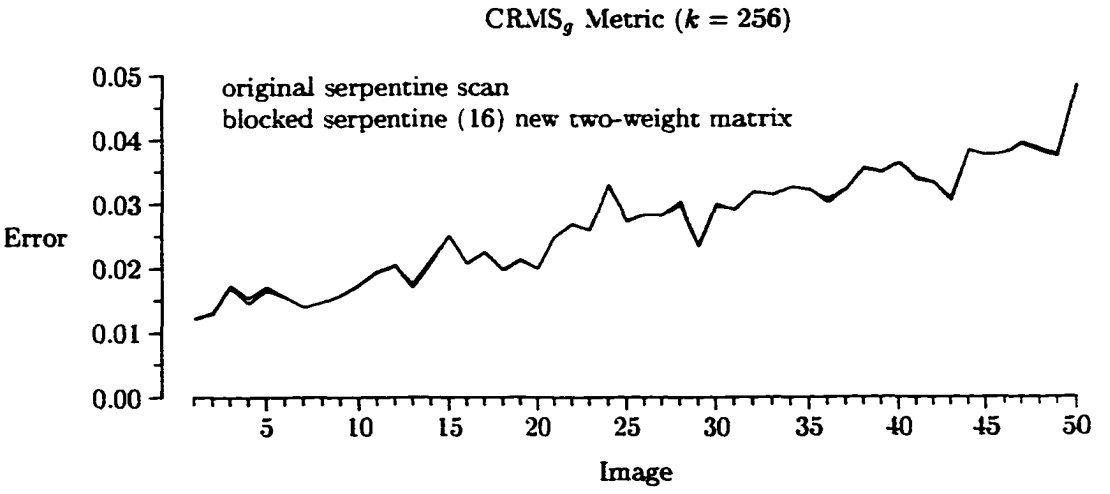


Figure 5.32: Comparing the original serpentine scan using the Floyd-Steinberg matrix with the tiled scan (block size = 16) using the new two-weight matrix.

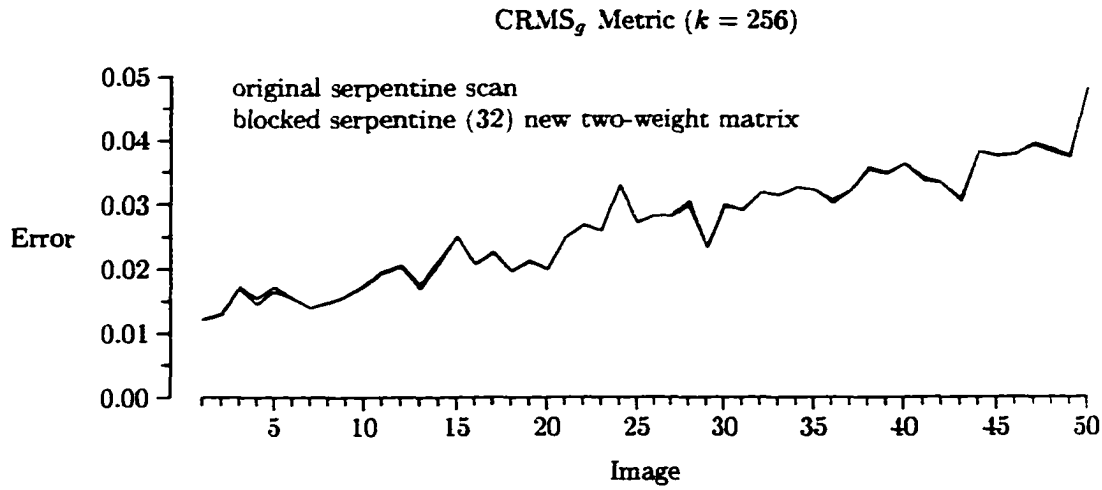


Figure 5.33: Comparing the original serpentine scan using the Floyd-Steinberg matrix with the tiled scan (block size = 32) using the new two-weight matrix.

Chapter 6

Conclusions and Future Work

Color image quantization is a very popular topic today given the rise of multimedia and the world wide web. The presentation of color images on devices with limited color capabilities requires a reduction in the number of colors contained in the images. This reduction is performed using a color image quantization algorithm. The desired result is an image that resembles the original as closely as possible. The quantization algorithm attempts to select k colors that best represent the contents of the image. The original image is then recolored using the representative colors. To improve the resulting image, a dithering process can be used in place of the recoloring.

Today, the quantization process is used in many applications where the results are needed in real-time. The ultimate goal in these situations is to produce an acceptable quantized image very quickly. Sometimes, the quality of the resulting image is compromised in favor of a fast result. In many instances, these fast results are visually unacceptable.

Given the current need for results in real-time, I have studied several aspects of the color quantization process. During this study, I developed new techniques that can be

applied to the color quantization process that improves the execution time while improving or maintaining the quality of the resulting image. I provide a review of these improvements in the following sections. I then conclude the chapter by presenting several topics that deserve further investigation.

6.1 The Color Cut Quantization Algorithm

I developed a new heuristic algorithm to solve the color quantization problem. My algorithm, which I call the color cut algorithm, is a variation of Heckbert's median cut algorithm. My algorithm sub divides the RGB color space by creating boxes containing approximately the same number of colors; each box represents an entry in the representative set.

My algorithm produced better visual results than the popular median cut and octree quantization algorithms. The measurement of the improvement was computed using both the relative root-mean-square and relative convolution root-mean-square error metrics. The results were displayed in several graphs of comparisons between the different algorithms.

In some instances, the construction of the partition can be done using the centroid mapping scheme as opposed to the nearest-neighbor mapping scheme. The centroid mapping scheme maps each input color to a representative based on the box (as described in Chapter 4) in which that color lies. The nearest-neighbor scheme is generally the preferred method since it produces better visual results. The centroid mapping scheme, however, is faster. Hence, I experimented with the use of a centroid mapping scheme in my color cut algorithm. While the visual results were not as good as those produced using the nearest-

neighbor scheme, they were still much better than those produced by the median-cut or octree algorithms.

The median cut and octree algorithms are very common because they can be used on computers with a limited amount of memory. To reduce the amount of memory required, the median cut algorithm performs a bit reduction by setting the three least significant bits of each color component to zero. The affect is to divide each color component by eight, thus reducing the maximum possible colors from 16.67 million to 32,768.

With few modifications to the data structures, I provided a method to include a bit reduction scheme in my color cut algorithm. I experimented with three different schemes: a 1-1-1 reduction, a 2-2-2 reduction, and a 3-3-3 reduction. When compared to the color cut algorithm with no reduction, there is little difference with a 1-1-1 reduction. There is a more significant difference, however, when using the 2-2-2 and 3-3-3 reduction schemes. When compared to the results of the median cut and octree algorithms, the color cut algorithm with a 3-3-3 reduction scheme still produces better results.

I analyzed the worst case time complexity of the color cut algorithm with no bit reduction to be $O(n \log k)$, where n is the number of pixel in the image. If bit reduction is used, the time complexity is $O(c \log k)$ where c is the number of unique colors after bit reduction. The time complexity of the bit reduction implementation is equal to that of the median cut algorithm. Both the bit reduction and non-bit reduction implementations have a better time complexity than the octree algorithm which has a time of $O(nk)$.

In addition to the theoretical worst case time, I also computed the empirical analysis of the algorithms using the set of 50 images described in Appendix C. The results from this testing showed that the running time of the octree algorithm is better than the color cut

algorithm with no bit reduction for approximately half the images and worst for the others. When compared to the median cut algorithm, the color cut algorithm with bit reduction is slightly faster.

Therefore, my color cut algorithm could be used to produce better visual results than the median cut and octree algorithms. If memory is limited or time is important, then a bit reduction scheme can be used; my algorithm still produces better results than the other algorithms. The running time is better than that of the median cut algorithm and approximately equal to that of the octree algorithm on average.

6.2 The Adaptive Locally Sorted Search Algorithm

After producing the representative color set, the original colors from the image have to be mapped to a color in the representative set. This mapping produces the partition. There are two methods that that can be used for this mapping: centroid mapping or nearest-neighbor mapping. The nearest-neighbor mapping is the most common approach. Thus, we need a fast method to perform the searches for a nearest-neighbor. The most popular method used in quantization is the locally sorted search (LSS) algorithm proposed by Heckbert.

The LSS algorithm subdivides the RGB color space into a number of smaller equal sized cubes. A list is then constructed for each cube, as needed, containing the representative colors that could be a possible nearest-neighbor of any color within that cube. This method reduces the number of representative colors that must be searched when mapping a single color. There is a drawback to this method, however. The division of the RGB color space

into equal sized cubes leave many of the cubes unused. Thus, there could be some lists containing a large number of eligible nearest-neighbors.

To improve the search time for a nearest-neighbor, I developed the adaptive locally sorted search (ALSS) algorithm which is based on the LSS algorithm. My algorithm subdivides the RGB color space into boxes of varying sizes depending upon the location of the representative colors. Search lists are then constructed in a manner similar to that of the LSS algorithm. Given the method of creating the boxes, I have shown a way to reduce the number of representative colors contained in the search lists used by the ALSS algorithm.

The time to construct the search lists in the ALSS algorithm was shown to be $O(k^2)$ where k is the number of representative colors. The construction of the LSS algorithm depends on the number of boxes created. Heckbert showed that his algorithm has a construction time of $O(d^3k+d^3L\log L)$ where d is the number of divisions along each dimension and L is the average list length.

After constructing the search list, the time to search for a nearest-neighbor is $O(k^2)$ in the worst case for both algorithms. Using empirical analysis, I was able to show that my ALSS algorithm reduces the search time as compared to the LSS by approximately 37 percent. Actual running times also show that the LSS algorithm is faster than the ALSS algorithm.

Thus, I propose the use of the ALSS algorithm for the nearest-neighbor searches required in the mapping phase of the quantization process. The implementation of my algorithm is no more complicated than that of the LSS algorithm.

6.3 Improvements in the Dithering Process

The image produced by the quantization process using a recoloring technique can contain artifacts such as contouring. This artifact can be reduced using the error-diffusion dithering technique to smooth the image by blending neighboring pixels. This blending fools the eye and brain into perceiving different shades of colors where none actually exists. This dithering process works by computing a difference value between the original color and the corresponding representative color as each pixel is processed. The difference is then distributed to the neighboring pixels that have not yet been processed. The error distribution is controlled by a weight matrix, the most popular of which is that due to Floyd and Steinberg.

The use of the error-diffusion technique with the Floyd-Steinberg weight matrix greatly improves most images when compared to the use of recoloring. I verified this using my new color cut quantization algorithm both with and without color reduction. The dithered results were much better than those produced by recoloring.

During this study, I discovered that a reduced weight matrix could produce results almost identical to that of the Floyd-Steinberg matrix. Instead of using four weights, my new matrix used only two, such that the error value is only distributed to the horizontal and vertical neighbors not yet processed. This reduction in the weight matrix can reduce the actual execution time by reducing the number of costly mathematical operations.

To verify the validity of claims concerning my new weight matrix, I conducted several experiments using the set of test images described in Appendix C. All of these experiments used my color cut quantization algorithm to construct the quantizer. First, I compared

the results of dithering with the Floyd-Steinberg matrix with those using my new matrix for both $k = 256$ and $k = 64$. There was little to no difference in both the visual and analytical results. Next, I used the color reduction schemes with the color cut quantization algorithm and performed the same experiments. Again, the dithered versions produced much better results than the recolored versions. The two-weight matrix can be used to reduce the execution time while producing almost identical results.

While error-diffusion is the most popular method for smoothing an image, it is not the only method for dithering an image. Two other popular techniques are ordered dithering and dot diffusion. Neither of these are well suited for reducing artifacts that result from the quantization process. However, both can be implemented in parallel. This is done by dividing the image into small blocks an then processing the blocks in parallel. Since error-diffusion requires the error to be distributed to pixels not yet processed, it has always been viewed as a sequential process.

In this thesis, however, I have shown that error-diffusion can be used in a parallel fashion. The approach taken is the same as for ordered-dithering and dot-diffusion; the image is divided into small equal sized blocks. The error-diffusion technique is then applied to each block independently. To determine the size of blocks into which the image should be subdivided, I conducted several experiments using block sizes of 8, 16, 32, 45, and 64. The results, based on both visual and analytical results showed little deviation from the sequential method. Only the use of an 8×8 block size showed noticeable degradation.

Based on the analytical and visual results, the error diffusion algorithm can be implemented in a parallel fashion. Though I do not provide that implementation, I have shown that the results produced from such an implementation would be similar to those produced

by the traditional serial approach. This blocking approach could also be used in a sequential fashion to reduce the amount of memory required to process an image. Thus, I term it error diffusion by tiling.

6.4 Future Work

First, I believe it should be investigated whether the search time of the ALSS algorithm can be improved by using some tree structure other than the kd-tree. For example, an octree or some variant of the binary or quadtree may produce better search times.

Second, the color reduction scheme used with the color cut algorithm provides another avenue for future study. In this thesis, I limited my work to the use of 1-1-1, 2-2-2, and 3-3-3 schemes. While these schemes produced good results, some other scheme such as a 3-2-4, 2-1-3, or 2-1-2 scheme may work even better.

The purpose of the color reduction scheme is to reduce the execution time of a quantization algorithm by reducing the number of colors that have to be processed. The selected reduction scheme is applied to each color in the image. Perhaps, however, the number of colors can be reduced by computing an average color over a small local area or neighborhood. This average color could then be used as an original input color. In the worst case, the number of colors could be reduced based on the size of the local area.

Third, my study of the dithering phase of the quantization process only dealt with display devices. It is well known that the presentation of an image on a display device differs from the presentation on a hardcopy device. Thus, further study is needed to test the validity of the new weight matrix and tiling approach for use on hardcopy devices.

Fourth, during my research, I found that some of the quantization algorithms work better than others in most cases. There are some situations, however, where this is not the case. This is especially true where a color reduction scheme is used. Thus, if an image could be analyzed before performing the quantization process to determine which algorithm or version of algorithm should be used, better results could be achieved.

Finally, I believe that some of my work could be used to improve the quantization process of video sequences. Some work has been done in this area, but more is needed. This is especially true in the reduction of the execution time of algorithms.

Appendix A

ALSS Theorem and Proof

My Adaptive Locally Sorted Search algorithm in Chapter 4 relies on the assertion that some possible nearest-neighbor candidates can be removed from contention if they lie outside a given region. The region was defined to be the union of the volumes of eight spheres centered at the eight vertices of a box and passing through a common point inside the box. In this appendix, I prove the two-dimensional version of that assumption.

The proof of the three-dimensional version is conceptually identical to the one given here. However, the sketches are much harder to draw and to decipher, and the number of parts in the proof of the analog of Theorem A.2 is greater than the number that appears here. Theorem A.1 is the primary result.

Theorem A.1 *Let R be a rectangle with vertices v_0, v_1, v_2, and v_3. Let y be a point on or inside R. For each i, where $i \in \{0, 1, 2, 3\}$, let C_i be the circle centered at v_i with radius yv_i. Let x be any point inside or on R and let p be a point which lies outside each of the circles C_0, C_1, C_2, and C_3. Then $d(x,p) > d(x,y)$, where $d(r,s) = \sqrt{(r_x - s_x)^2 + (r_y - s_x)^2}$.*

The diagram in Figure A.1 illustrates the configuration for Theorem A.1. The proof of the theorem rests on the following four lemmas and one theorem. The lemmas are results from elementary geometry; their proofs are omitted here.

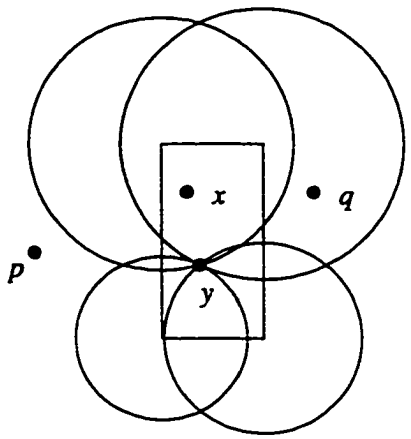


Figure A.1: Illustration of the rectangle and circles for Theorem A.1.

Lemma A.1 *Let D be a circle centered at x and passing through y. Let p be a point which lies outside D. Then $d(x,p) > d(x,y)$.*

Lemma A.2 *Let C be a circle centered at v and passing through y. Let O be a circle centered at x and passing through y. Suppose that x lies on the line yv. Then O lies entirely inside C.*

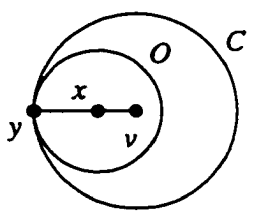


Figure A.2: Illustration of a circle that is internally tangent to a second circle.

Lemma A.3 *Suppose that v_0, v_1, and y are three non-collinear points. Let C_0 be a circle centered at v_0 and passing through y. Let C_1 be a circle centered at v_1 and passing through y. Then C_0 and C_1 intersect at two distinct points.*

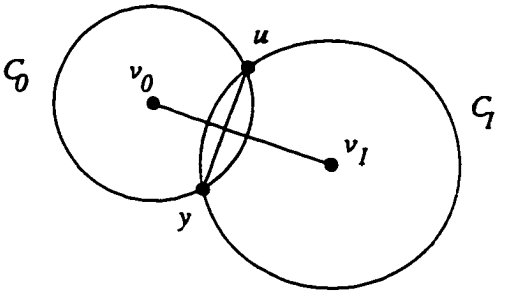


Figure A.3: Illustration of a circle with intersections at two distinct points (Lemma A.3) and a perpendicular bisector (Lemma A.4).

Lemma A.4 *Let C_0 be a circle centered at v_0. Let C_1 be a circle centered at v_1. Suppose that C_0 and C_1 intersect in the distinct points y and u. Then v_0v_1 is the perpendicular bisector of yu.*

Theorem A.2 *Let T be a triangle with non-collinear vertices v_0, v_1, and y. Let C_0 be a circle centered at v_0 and passing through y. Let C_1 be a circle centered at v_1 and passing through y. Let x lie inside or on the triangle T, and let p be a point which lies outside each of circles C_0 and C_1. Then $d(x,p) > d(x,y)$.*

Proof: Let O be a circle centered at x and passing through y. It suffices to show that O lies entirely inside or on the union of the circles C_0 and C_1. Suppose it does. Since p lies outside each of C_0 and C_1, it surely lies outside O. So Lemma A.1 assures that $d(x,p) > d(x,y)$.

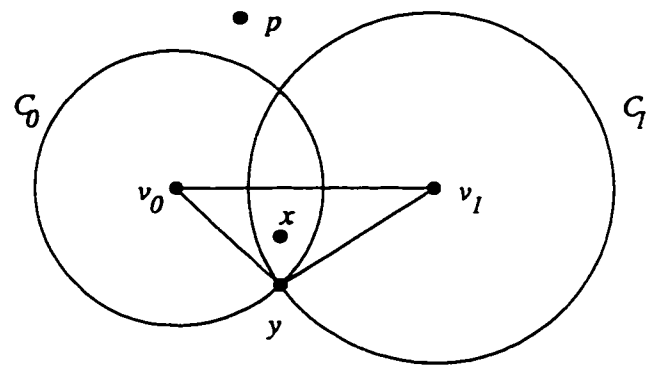


Figure A.4: Illustration of the triangle and circles from Theorem A.2.

The remainder of the proof consists of four parts.

Part 1. If x lies on yv_0 then O lies entirely inside the required union.

Part 2. If x lies on yv_1 then O lies entirely inside the required union.

Part 3. If x lies on v_0v_1 then O lies entirely inside the required union.

Part 4. If x lies inside T then O lies entirely inside the required union.

Proof of Part 1: By Lemma A.2, O lies entirely inside C_0 and so it lies inside the union of C_0 and C_1.

Proof of Part 2: The proof of Part 2 is like the proof of Part 1.

Proof of Part 3: Lemmas A.3 and A.4 assures us that v_0v_1 is the perpendicular bisector of uy. So any circle drawn with its center on v_0v_1 and passing through y will also pass through u. (The symmetries are clear in Figure A.5.) Our O is such a circle and so it lies inside the required union.

Proof of Part 4: Let b be the intersection of the line containing both y and x with v_0v_1. Let E be the circle centered at b passing through y. See Figure A.6. Part 3 of this proof assures that E lies entirely inside the required union. Lemma A.1 assures that O lies entirely inside E. So O lies entirely inside the required union. ■

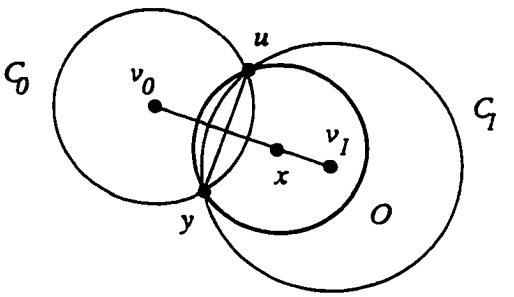


Figure A.5: Illustration of the three circles passing through two common points.

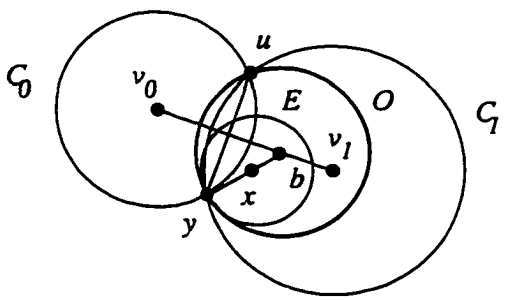


Figure A.6: Illustration of the intersection of a circle centered at a point inside the triangle.

Proof of Theorem A.1: For each i, where $i \in \{0, 1, 2, 3\}$, draw yv_i thus partitioning R into four triangles. Our point x lies inside or on at least one of these. Select one of these and call it T. Apply Theorem A.2, with appropriate renaming of the subscripts in v_0, v_1, C_0, and C_1, for the required conclusion. ■

The three-dimensional version of Theorem A.1 is interpreted to assure us that any point in a box is closer to the representative color, y, in that box than it is to any representative color outside the spheres centered at the vertices of the box, passing through y. Thus, only the representative colors that lie inside one or another of the spheres can possibly be a nearest-neighbor of a color inside the box. Those colors lying outside all of the spheres defined in the theorem can be removed from further consideration. In practice, this will reduce the number of comparisons required to find the nearest-neighbor of a given color.

Appendix B

Dithering Algorithm Implementations

This appendix provides the code for an implementation of the Floyd-Stienberg dithering technique for use on color display devices. The code below is a modified version of the code written by Thomas G. Lane [42] for use with the JPEG image format library. Its most significant feature is the lack of floating point arithmetic. Instead of distributing the error to the actual pixel values in the image array, the error values are accumulated in an error buffer. When a pixel is processed, the corresponding error value is divided by 16. It is then clamped to the appropriate value and added, component by component, to that pixel's color. The clamping is performed using a look up table which is filled in a preprocessing phase.

The error buffer need only be a 1-d array with each element being a structure containing three fields to store the error values. The length of the array is the width of the image plus

two. These extra two elements allow for the distribution of the error to the pixel below and back of the first pixel on the row and to the pixel below and after the last pixel.

B.1 The Floyd-Stienberg Technique

The function that implements the Floyd-Stienberg dithering technique requires three pieces of data: (1) the image to be dithered which will also be used to store the dithered results, (2) the width of the image, and (3) the height of the image. The prototype of the function is as follows

```
void fsDither( Image I, int width, int height )
```

The data type **Image** is assumed to be a two-dimensional array of the given width and height with each cell containing a byte-triple representing a 24-bit color.

This implementation attempts to reduce the number of comparisons by only using loops when absolutely necessary. Thus, many of the component by component operations are performed individually instead of within the confines of a loop. In the following variable declaration, the data type **Error** is assumed to be a structure containing three short integer fields that are used to store the error values.

```
rgb  color;      // color from the original image.
rgb  repcolor;   // representative color to which color maps.
byte *pix;       // pointer to current image pixel within row.
short  cTable[511]; // actual error clamping buffer: -255..255.
short *clamp;       // points to the middle entry of cTable.

Error *errorBuf;  // error buffer. Holds values for next row.
Error *errPtr;    // current position in the error buffer.
```

```
Error curErr;    // the current error and the error to the side.
Error prevErr;   // error for the element below and back.
Error downErr;   // error for the element below the current.

int dir = 1;       // error pointer increment/decrement.
int dir3 = 3;      // image pointer increment/decrement.
int evenRow = 1;   // is this an even or odd row?
```

Before processing the image, the error buffer and error clamping table needs to be constructed and initialized. The use of the clamping table depends on the fact that array subscripts in C++ can be negative to reference elements before the current pointer position.

```
    /* Allocate space for the error buffer. */
errorBuf = new Error [ (width+2) ];

    /* Initialize the buffer to zero. */
memset( errorBuf, 0, (width+2) * sizeof( Error ) );

    /* Build the error clamping table. */
clamp = BuildClampTable( cTable );
```

The `BuildClampTable()` function fills in the elements of the clamping table. A pointer to the center element is returned. This pointer can then be referenced with either positive or negative indicies to access the appropriate value.

```
short * BuildClampTable( short *Table )
{
   int ndx, value = 0;

   Table[0] = value++;
   for( ndx = 1; ndx < 16; ndx++, value++ ) {
      Table[ndx] = value;
      Table[-ndx] = -value;
   }
   for( ; ndx < 48; ndx++, value += (ndx & 0x01) ) {
      Table[ndx] = value;
      Table[-ndx] = -value;
   }
   for( ; ndx < 256; ndx++ ) {
      Table[ndx] = value;
      Table[-ndx] = -value;
   }
   return( &Table[256] );
}
```

Now comes the actual dithering process. All of the code in the following paragraphs is performed within a loop that is executed **height** times starting at zero with a loop variable of **y**. In processing each row, a pointer (**ppix**) is set to the first or last element of the current image row depending upon whether it is an even or odd row. On an even row, as the image is processed, the pointer is incremented across the row; on odd rows, it is decremented.

To reduce the number of computations, a flag (**evenRow**) is used to keep track of even and odd rows. In addition, two variables are used to indicate whether the image pointer (dir3) and error buffer pointer (dir) are incremented or decremented. Two variables are needed since the image pointer needs to be changed by three elements (bytes) while the error pointer only needs to be changed by one element.

```
if( evenRow ) {   // if this is an even row.
   errPtr = errorBuf;
   ppix = &I[y][0];
   dir = 1;
   dir3 = 3;
   evenRow = 0;
}
else {
   errPtr = errorBuf + (width+1);
   ppix = &I[y][width-1];
   dir = -1;
   dir3 = -3;
   evenRow = 1;
}

    /* Initialize the error values. */
curErr.r = curErr.g = curErr.b = 0;
prevErr = downErr = curErr;
```

With the completion of the initialization, the actual row processing occurs. The following parts are executed for each pixel on the current row. The first step is to compute the error that was propagated to the current pixel, clamp it and then add it to the pixel's color. After adding the error, the color components must be checked and clamped to the range $[-255 \ldots 255]$.

```
    /* We divide by 16 and round at this point. */
curErr.r = (curErr.r + errPtr[dir].r + 8 ) / 16;
curErr.g = (curErr.g + errPtr[dir].g + 8 ) / 16;
curErr.b = (curErr.b + errPtr[dir].b + 8 ) / 16;

    /* Check the error for this pixel and clamp it. */
curErr.r = clamp[ curErr.r ];
curErr.g = clamp[ curErr.g ];
curErr.b = clamp[ curErr.b ];
```

```
    /* Add error to the pixel and clamp to 255. */
int temp = ppix[0] + curErr.r;
color.r = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;
temp = ppix[1] + curErr.g;
color.g = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;
temp = ppix[2] + curErr.r;
color.b = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;
```

Next, the representative color or nearest neighbor of the pixel's adjusted color is found using a nearest neighbor search routine, `q()`. The pixel is then recolored with the representative color. The difference between the original color and the representative color is then computed, component by component. These values are then distributed to the neighboring pixels using the temporary error variables.

```
    /* Search for the color's nearest neighbor. */
repcolor = q( color );

    /* Recolor the current pixel. */
ppix[0] = repcolor.r;
ppix[1] = repcolor.g;
ppix[2] = repcolor.b;

    /* Compute the difference error. */
curErr.r = color.r - (int)repcolor.r;
curErr.g = color.g - (int)repcolor.g;
curErr.b = color.b - (int)repcolor.b;

    /* Distribute to the pixel below and left. */
errPtr->r = prevErr.r + curErr.r * 3;
errPtr->g = prevErr.g + curErr.g * 3;
errPtr->b = prevErr.b + curErr.b * 3;

    /* Distribute to the pixel below. */
prevErr.r = downErr.r + curErr.r * 5;
prevErr.g = downErr.g + curErr.g * 5;
prevErr.b = downErr.b + curErr.b * 5;
```

```
    /* Distribute to the pixel below and right. */
downErr = curErr;

    /* Distribute to the pixel to the right. */
curErr.r *= 7;
curErr.g *= 7;
curErr.b *= 7;
```

To conclude the processing of the current pixel, the error buffer pointer and current pixel pointers are adjusted to point to the next pixel.

```
errPtr += dir;
ppix += dir3;
```

After the current row is processed, the last entry of the error buffer needs to be set with the values from the prevErr variable: `*errPtr = prevErr;`. After processing the entire image, the error buffer is destroyed.

B.2 Implementation of my Two-Weight Matrix

The implementation of the error diffusion dithering technique using my new two-weight matrix is provided below. It is derived from the implementation of the algorithm for the Floyd-Steinberg weight matrix described above.

```
void twoWeightDither( Image I, int width, int height )
{
   rgb   color;     // color from the original image.
   rgb  repcolor;   // representative color to which color maps.
   byte *pix;       // pointer to current image pixel within row.

   short  cTable[511]; // actual error clamping buffer: -255..255.
   short *clamp;       // points to the middle entry of cBuffer.
```

```
Error *errorBuf;  // error buffer. Holds values for next row.
Error *errPtr;    // current position in the error buffer.
Error curErr;     // the current error.

int dir = 1;      // error pointer increment/decrement.
int dir3 = 3;     // image pointer increment/decrement.
int evenRow = 1;  // is this an even or odd row?

   /* Allocate space for the error buffer. */
errorBuf = new Error [ width ];

   /* Initialize the buffer to zero. */
memset( errorBuf, 0, width * sizeof( Error ) );

   /* Build the error clamping table. */
clamp = BuildClampTable( cTable );

   /* Process and dither the image. */
for( int y = 0; y < height; y++ ) {
  if( evenRow ) {   // if this is an even row.
    errPtr = errorBuf;
    ppix = &I[y][0];
    dir = 1;
    dir3 = 3;
    evenRow = 0;
  }
  else {
    errPtr = errorBuf + (width-1);
    ppix = &I[y][width-1];
    dir = -1;
    dir3 = -3;
    evenRow = 1;
  }

     /* Initialize the error values. */
  curErr.r = curErr.g = curErr.b = 0;
```

```
        /* Process the current row. */
      for( int x = 0; x < height; x++ ) {

          /* We divide by 2 and round at this point. */
        curErr.r = (curErr.r + errPtr[dir].r + 1 ) >> 2;
        curErr.g = (curErr.g + errPtr[dir].g + 1 ) >> 2;
        curErr.b = (curErr.b + errPtr[dir].b + 1 ) >> 2;

          /* Check the error and clamp it. */
        curErr.r = clamp[ curErr.r ];
        curErr.g = clamp[ curErr.g ];
        curErr.b = clamp[ curErr.b ];

          /* Add error to the pixel and clamp to 255. */
        int temp = ppix[0] + curErr.r;
        color.r = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;
        temp = ppix[1] + curErr.g;
        color.g = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;
        temp = ppix[2] + curErr.r;
        color.b = (temp > 255) ? 255 : (temp < 0) ? 0 : temp;

          /* Search for the color's nearest neighbor. */
        repcolor = q( color );

          /* Recolor the current pixel. */
        ppix[0] = repcolor.r;
        ppix[1] = repcolor.g;
        ppix[2] = repcolor.b;

          /* Compute the difference error. */
        curErr.r = color.r - (int)repcolor.r;
        curErr.g = color.g - (int)repcolor.g;
        curErr.b = color.b - (int)repcolor.b;

          /* Distribute to the pixel below. */
        *errPtr = curErr;
        errPtr += dir;
        ppix += dir3;
      } // for( x )
    } // for( y )

    delete [] errorBuf;
  } // twoWeightDither()
```

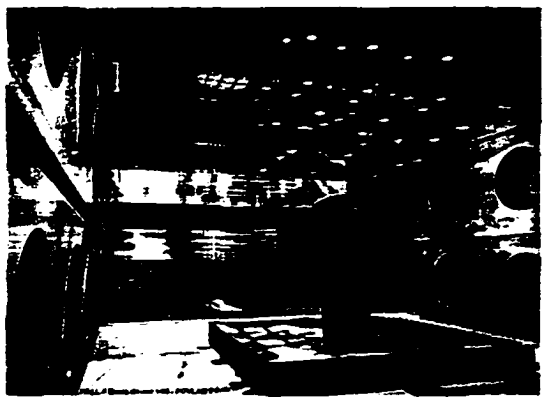
Appendix C

Set of Test Images

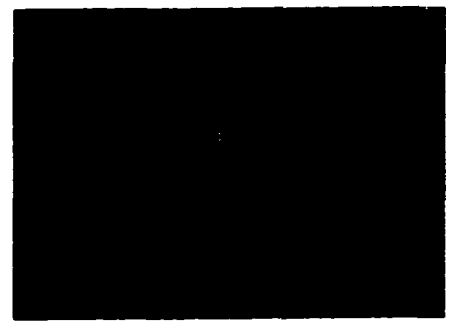
My research required the use of color images for comparing the results of applying various image manipulation algorithms. I wanted to choose a number of images so I could draw conclusions about my own contributions. It is common in the literature to draw conclusions based on the analysis of results from two or three images. To present my results with some impartiality, I have opted to work with fifty images. I selected images that are accessible to anyone with internet access. Thus, any of the experiments described in this thesis can be easily reproduced. The images come from the POV ray-tracing package [75] since ray-traced images commonly contain numerous colors and shapes, such as spheres, that produce good evaluation images.

The set of images that I choose for my experiments are shown on the following pages. Each image was converted from their native format to Postscript (tm) and reduced to one fourth their original size. For each image, four pieces of information is provided: (1) the name of the image as it is identified in the POV package, (2) the number of colors contained

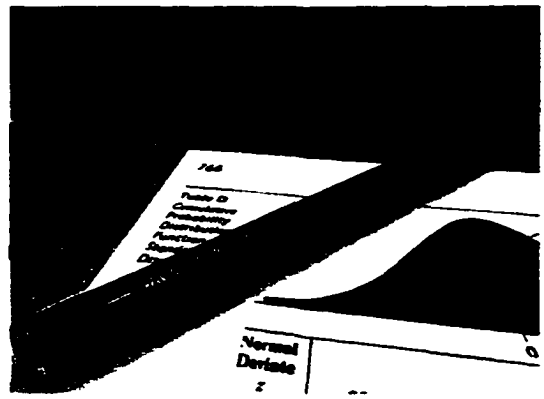
in the image, (3) the size (width × height) of the image in pixels, and (4) the image number used in the graphs throughout the thesis.



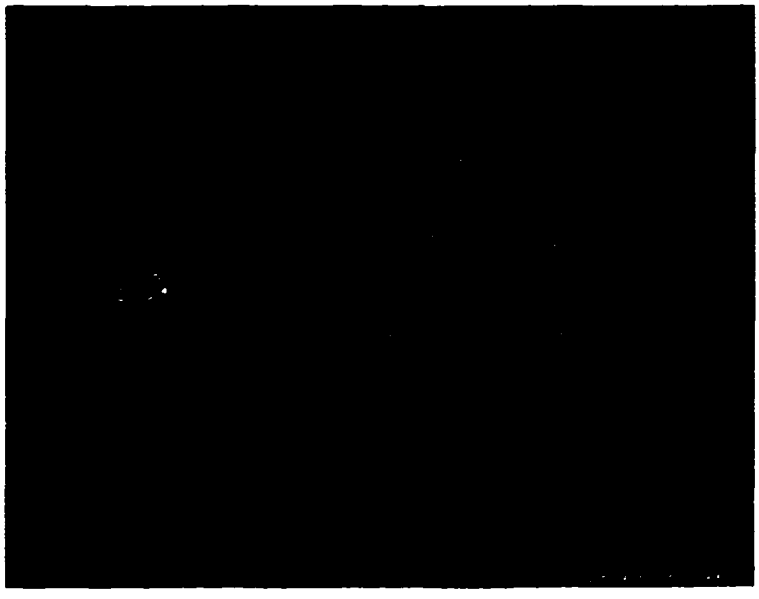
(1) hall - 22180 (800 x 600)



(2) NewPlanet - 17128 (640 x 480)

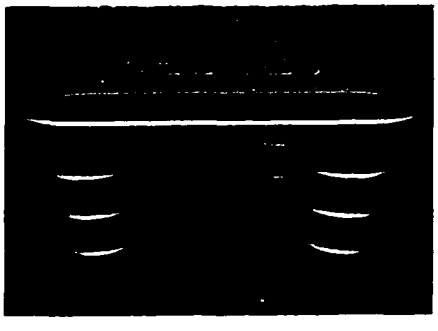


(3) math - 60063 (800 x 600)



(4) crest1 - 23243 (1125 x 900)

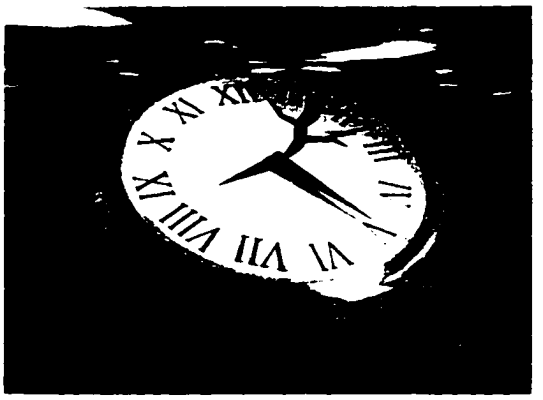
(5) cboxray - 11146 (640 x 480)



(6) ikea2020 - 23246 (640 x 480)

(7) primitiv - 36676 (640 x 480)

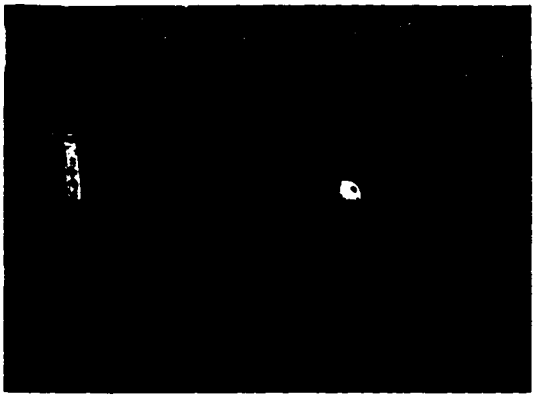
(8) sag - 6642 (640 x 480)



(9) fractim2 - 62595 (800 x 600)

(10) mons - 20687 (640 x 480)

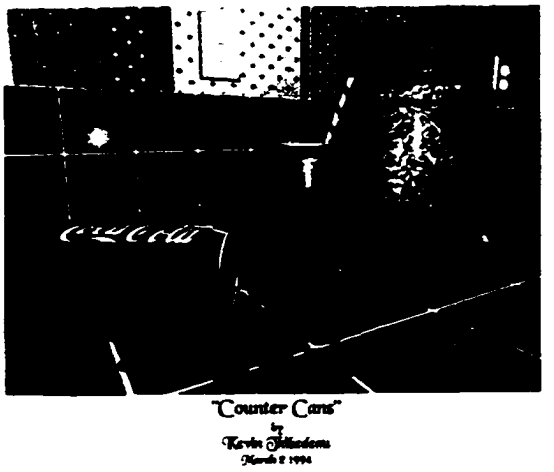
(11) cannon - 35744 (1125 x 900)



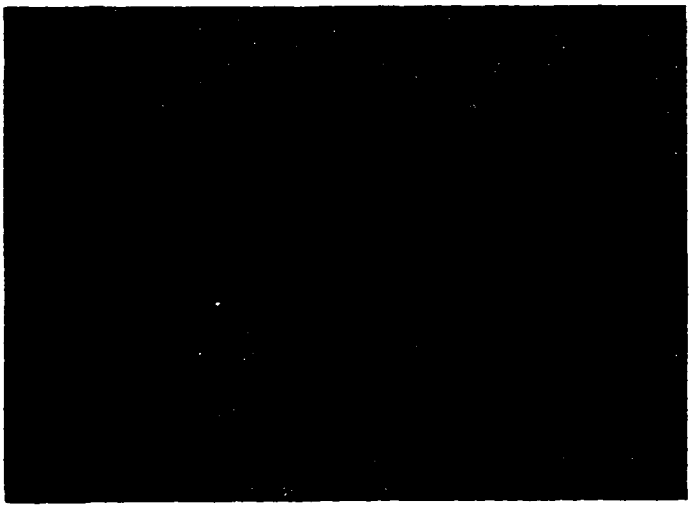
(12) pool - 38115 (800 x 600)

(13) glasssph - 34745 (640 x 480)

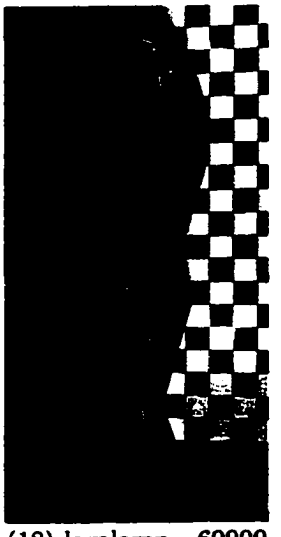
(14) roman - 55860 (800 x 600)



(17) cokecans - 108991 (800 x 720)

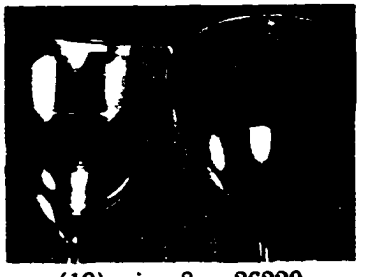


(15) china - 39379 (1024 x 768)

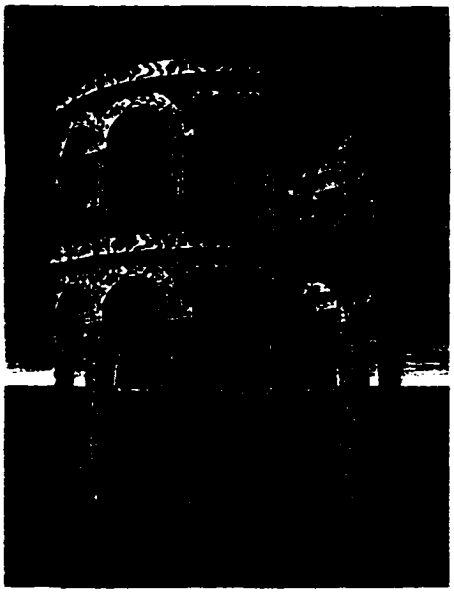


(18) lavalamp - 69999 (400 x 800)

(16) piano5gc - 77034 (1000 x 640)



(19) wines8e - 26229 (529 x 398)

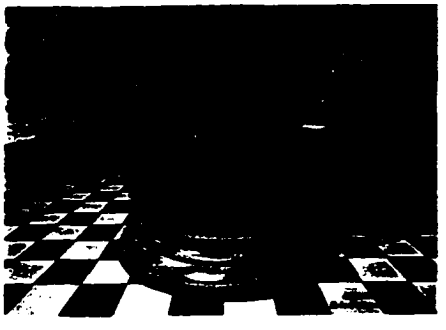


(20) DVinci02 - 94229 (675 x 900)

(22) globe - 12354
(480 x 360)

(23) camera1a - 53964 (640 x 480)

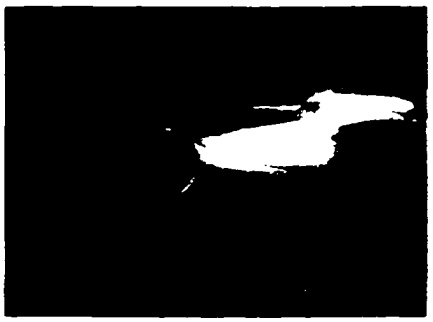
(21) mindroom - 86984 (800 x 600)



(24) belljar - 18437 (640 x 480)

(25) mndawn - 73423 (1024 x 768)

(26) cut5 - 65637
(480 x 640)

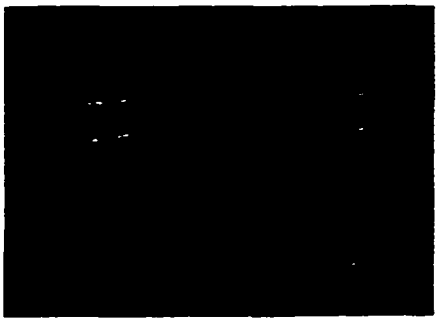


(27) JM-SuperNova - 65028
(640 x 480)

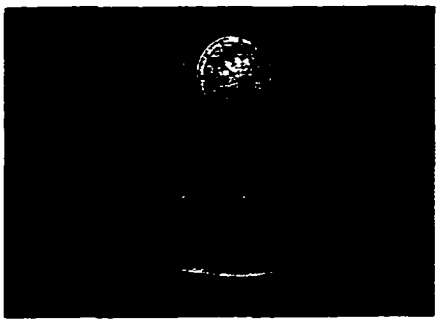
(28) pot9 - 36302
(528 x 396)

(29) paprwgt - 43902 (640 x 480)

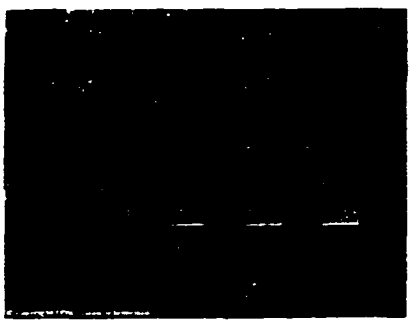
(30) faceless - 32361 (800 x 600)



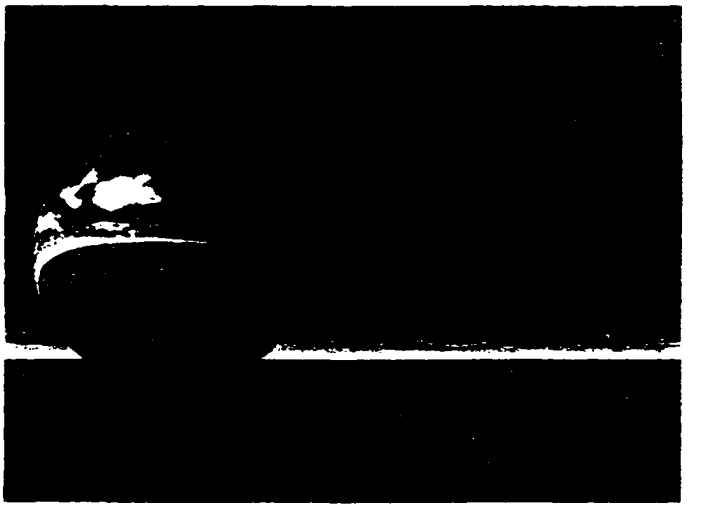
(31) juice - 59357 (640 x 480)



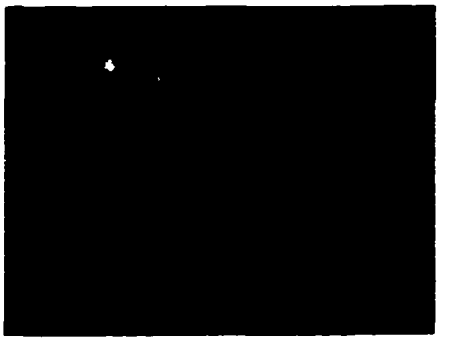
(32) mantel - 47125 (640 x 480)



(33) chess - 67933 (600 x 480)

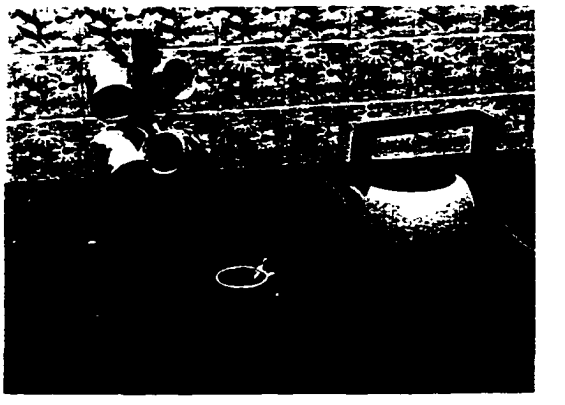


(34) DVinci01 - 64558 (1024 x 768)

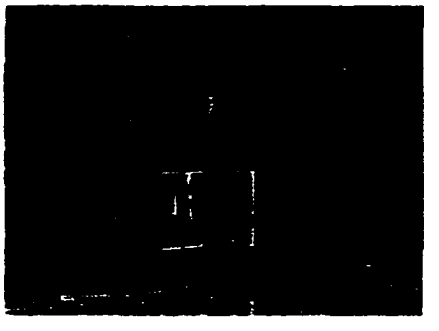


(37) lethel - 53048 (643 x 509)

(35) dragons - 87518 (800 x 600)



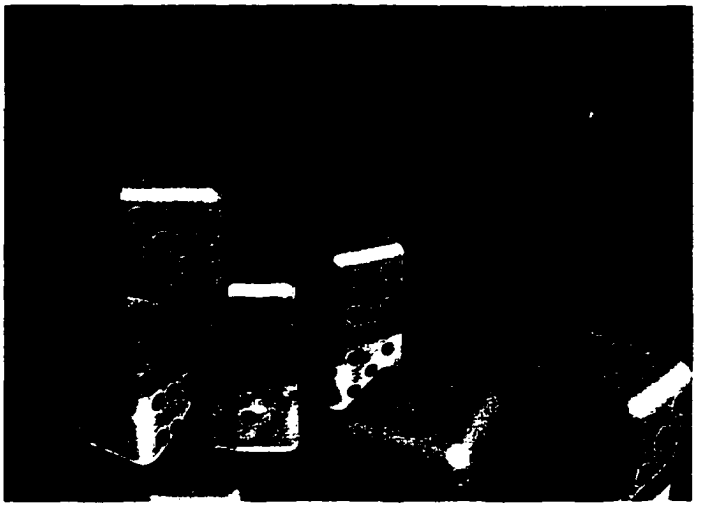
(38) kitchrea - 134832 (800 x 600)



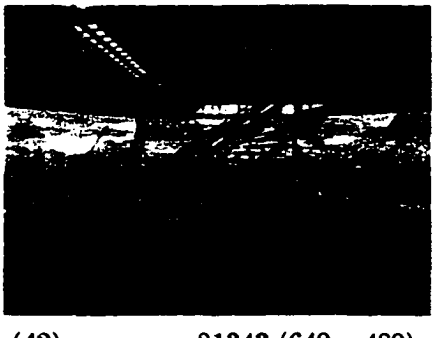
(36) fireplce - 51739 (640 x 480)

(39) JM-Prtal - 75799 (800 x 600)

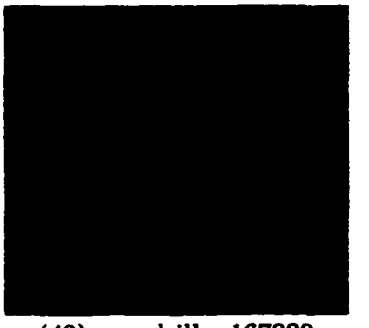
(40) mrtiens4 - 36508 (800 x 600)



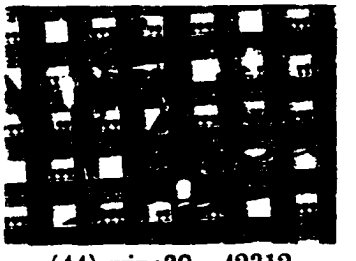
(41) diceball - 123349 (1024 x 768)



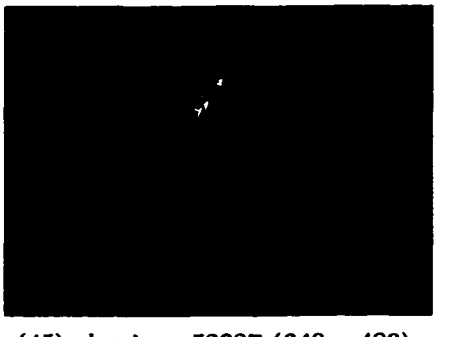
(42) museum - 81343 (640 x 480)



(43) mandrill - 167990
(512 x 480)



(44) wine39 - 42312
(496 x 372)



(45) plntrise - 52387 (640 x 480)

(46) blessed - 56625 (640 x 512)

(47) island2 - 176545 (1024 x 768)

(48) caustic1 - 95990 (640 x 480)

(49) herald - 97739 (800 x 600)

(50) paintjar - 53073 (600 x 480)

Bibliography

[1] D. P. ANDERSON. Techniques for reducing pen plotting time. *ACM Transactions on Graphics*, 2(3):197–212, July 1983.

[2] MICHAEL A. ANDREOTTOLA. Color hard-copy devices. In *Color and the Computer*. H. John Durrett, editor, volume 1, chapter 12, pages 221–240. APRESS, New York, 1987.

[3] EDWARD ANGEL. *Computer Graphics*. Addison-Wesley Publishing Co., 1990.

[4] RAJA BALASUBRAMANIAN AND JAN P. ALLEBACH. A new approach to palette selection for color images. *Journal of Imaging Technology*, 17(6):284–290, December 1991.

[5] RAJA BALASUBRAMANIAN, JAN P. ALLEBACH, AND CHARLES A. BOUMAN. Color-image quantization with use of a fast binary splitting technique. *Journal of the Optical Society of America*, 11(11):2777–2786, November 1994.

[6] B. E. BAYER. An optimum method for two-level rendition of continuous-tone pictures. *Conference Record of the International Conference on Communications*, 26:26–11 – 26–15, 1973.

[7] JON LOUIS BENTLEY. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[8] F. BILLMEYER AND M. SALTZMAN. *Principles of Color Technology*. Wiley, New York, second edition, 1981.

[9] JOHN BRADLEY. xv: Interactive image display for the x window system. A software application, 1994. Version 3.10a. Includes code from the Independent Joint Photographic Experts Group (JPEG).

[10] DENNIS BRAGG. A simple color reduction filter. In *Graphics Gems*, David Kirk, editor, volume 3, chapter I.4, pages 20–22. Academic Press, San Diego, CA, 1992.

[11] P. BRUCKER. On the complexity of clustering problems. In *Optimization and Operations Research*, R. Henn, B. Korte, and W. Oettly, editors, pages 45–54. Springer-Verlag, New York, 1977.

[12] GRAHAM CAMPBELL, THOMAS DEFANTI, JEFF FREDERIKSEN, STEPHEN JOYCE, LAWRENCE LESKE, JOHN LINDBERG, AND DANIEL SANDIN. Two bit/pixel full color encoding. *ACM Computer Graphics*, 20(4):215–233, August 1986.

[13] THOMAS H. CORMEN, CHARLES E. LEISERSON, AND RONALD L. RIVEST. *Introduction to Algorithms.* The MIT Press, Cambridge MA, 1990.

[14] T. N. CORNSWEET. *Visual Perception.* APRESS, New York, 1970.

[15] PAMELA C. COSMAN, KAREN L. OEHLER, EVE A. RISKIN, AND ROBERT M. GRAY. Using vector quantization for image processing. *Proceedings of the IEEE*, 81(9):1326–1341, September 1993.

[16] H. DE RIDDER. Minkowski metrics as a combination rule for digital image impairments. *SPIE Proceedings*, 1666:16–26, 1992.

[17] SUDHIR S. DIXIT. Quantization of color images for display/printing on limited color output devices. *Computers and Graphics*, 15(4):561–567, 1991.

[18] RAPHAEL A. FINKEL AND JON L. BENTLEY. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, January 1974.

[19] R. W. FLOYD AND L. STEINBERG. An adaptive algorithm for spatial gray scale. *Society for Information Display*, 36:36–37, 1975.

[20] JAMES FOLEY, ANDRIES VAN DAM, STEVEN FEINER, AND JOHN HUGHES. *Computer Graphics: Principles and Practices.* Addison-Wesley Publishing Co., 2nd edition, 1987.

[21] JEROME H. FRIEDMAN, JON LOUIS BENTLEY, AND RAPHAEL ARI FINKEL. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[22] HENRY FUCHS, ZVI M. KEDEM, AND BRUCE F. NAYLOR. On visible surface generation by a priori tree structure. *ACM Computer Graphics*, 14(3):124–133, July 1980.

[23] G. Garcia and I. Herman, editors. *Advances in Computer Graphics VI.* Springer-Verlag, 1991.

[24] M. R. GAREY, D. S. JOHNSON, AND H. S. WITSENHAUSEN. The complexity of the generalized lloyd-max problem. *IEEE Transactions on Information Theory*, 28(2):84–95, March 1982.

[25] RONALD S. GENTILE, JAN P. ALLEBACH, AND ERIC. WALOWIT. Quantization of color images based on uniform color spaces. *Journal of Imaging Technology*, 16(1):11–21, February 1990.

[26] RONALD S. GENTILE, ERIC WALOWIT, AND JAN P. ALLEBACK. Quantization and multilevel halftoning of color images for near-original image quality. *Journal of the Optical Society of America*, 7(6):1019–1026, June 1990.

[27] ALLEN GERSHO. Principles of quantization. *IEEE Transactions on Circuits and Systems*, pages 427–436, 1978.

[28] ALLEN GERSHO AND ROBERT M. GRAY. *Vector quantization and signal compression.* Kluwer Academic Publishers, 1992.

[29] MICHAEL GERVAUTZ AND WERNER PURGATHOFER. A simple method for color quantization: Octree quantization. In *New Trends in Computer Graphics*, N. Magnenat-Thalmann and D. Thalmann, editors, pages 217–224. Springer-Verlag, Berlin, 1988.

[30] MICHAEL GERVAUTZ AND WERNER PURGATHOFER. A simple method for color quantization: Octree quantization. In *Graphics Gems*, Andrew S. Glassner, editor, volume 1, chapter 4, pages 287–293. Academic Press Professional, New York, NY, 1990.

[31] NAFTALY GOLDBERG. Colour image quantization for high resolution graphics display. *Image and Vision Computing*, 9(5):303–312, October 1991.

[32] ERNEST L. HALL. *Computer Image Processing and Recognition.* Academic Press, New York, NY, 1979.

[33] ROY HALL. *Illumination and Color in Computer Generated Imagery.* Springer-Verlag, New York, NY, 1989.

[34] STEPHEN HAWLEY. Ordered dithering. In *Graphics Gems*, Andrew S. Glassner, editor, volume 1, chapter 3, pages 176–178. Academic Press Professional, New York, NY, 1990.

[35] PAUL HECKBERT. Color image quantization for frame buffer display. *ACM Computer Graphics*, 16(3):297–307, July 1982.

[36] T. M. HOLLADAY. An optimum algorithm for halftone generation for displays and hard copies. *Proceedings of the Society for Information Display*, 21(2):185–192, 1980.

[37] YUANG-CHEN HSUEH, MING-GUEY CHERN, AND CHYI-HWA CHU. Image requantization by cardinality distribution. *Computers and Graphics*, 15(3):397–405, 1991.

[38] L. M. HURVICH AND D. JAMESON. *The Perception of Brightness and Darkness.* Allyn and Bacon, Boston, MA, 1966. pp 7–9.

[39] J. F. JARVIS, C. N. JUDICE, AND W. H. NINKE. A survey of techniques for the display of continuous tone pictures on bilevel displays. *Computer Vision, Graphics and Image Processing*, 5(1):13–40, March 1976.

[40] N. JAYANT, J. JOHNSTON, AND R. SAFRANEK. Signal compression based on models of human perception. *Proceedings of the IEE*, 81:1385–1422, October 1993.

[41] GREGORY JOY AND ZHIGANG XIANG. Center-cut for color-image quantization. *The Visual Computer*, 10(1):62–66, October 1993.

[42] Joint photographic experts group, 1994.

[43] JAMES M. KASSON AND WIL PLOUFFE. An analysis of selected computer interchange color spaces. *ACM Transactions on Graphics*, 11(4):373–405, October 1992.

[44] DONALD E. KNUTH. Digital halftones by dot diffusion. *ACM Transactions on Graphics*, 6(4):245–273, October 1987.

[45] ANTON KRUGER. Median-cut color quantization. *Dr. Dobb's Journal*, (219):46–54, September 1994.

[46] M. G. LAMMING AND W. L. RHODES. A simple method for improved color printing of monitor images. *ACM Transactions on Graphics*, 9(4):345–375, October 1990.

[47] E. H. LAND. Experiments in color vision. *Scientific American*, 200(5):84–99, May 1959.

[48] D. T. LEE AND C. K. WONG. Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[49] N. W. LEWIS AND J. A. ALLNATT. Subjective quality of television pictures with multiple impairments. *Electronic Letters*, 1:187–188, 1965.

[50] J. O. LIMB. Distortion criteria of the human viewer. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-9(12), December 1979.

[51] J. O. LIMB, C. B. RUBINSTEIN, AND J. E. THOMPSON. Digital coding of color video signals – a review. *ACM Transactions on Communications*, COM-25(11):1349–1385, November 1977.

[52] YOSEPH LINDE, ANDRES BUZO, AND ROBERT M. GRAY. An algorithm for vector quantizer design. *IEEE Transactions on Communications and Technology*, 28(1):84–95, January 1980.

[53] J. O. LIPPEL AND M. KURLAND. The effect of dither on luminance quantization of pictures. *IEEE Transactions on Communications and Technology*, 19(4):879–888, 1971.

[54] STUART P. LLOYD. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, IT-28(2):129–137, March 1982. Material in this paper presented at the Institute of Mathematical Statistics Meeting, Atlantic City, NJ, September 1957.

[55] FU-SHENG LU AND GARY L. WISE. A futher investigation of max'x algorithm for optimum quantization. *IEEE Transactions on Communications*, COM-33(7):746–750, July 1985.

[56] D. L. MACADAM. Uniform color scale. *Journal of the Optical Society of America*, (64):1961, 1974.

[57] BLAIR MACINTYRE AND WILLIAM B. COWAN. A pratical approach to calculating luminance contrast on a crt. *ACM Transactions on Graphics*, 11(4):336–347, October 1992.

[58] JAMES. L. MANNOS AND DAVID. J. SAKRISON. The effects of a visual fidelity criterion on the encoding of images. *IEEE Transactions on Information Theory*, IT-20(4):525–436, July 1974.

[59] JOEL MAX. Quantizing for minimum distortion. *IEEE Transactions on Information Theory*, IT-6:7–12, March 1960.

[60] NASIR D. MEMON AND KHALID SAYOOD. Lossless compression of rgb color images. *Optical Engineering*, 34(6):1711–1717, June 1995.

[61] ROBIN M. MERRIFIELD. Visual parameters for color crts. In *Color and the Computer*, H. John Durrett, editor, volume 1, chapter 3, pages 63–81. APRESS, New York, 1987.

[62] P. MERTZ, A.D. FOWLER, AND H. N. CHRISTOPER. Quality rating of television images. *Proceedings of the IRE*, 38(11):1269–1283, November 1950.

[63] GARY W. MEYER AND DONALD P. GREENBERG. Perceptual color spaces for computer graphics. In *Color and the Computer*, H. John Durrett, editor, volume 1, chapter 4, pages 83–100. APRESS, New York, 1987.

[64] GERALD MURCH. Color displays and color science. In *Color and the Computer*, H. John Durrett, editor, volume 1, chapter 1, pages 1–26. APRESS, New York, 1987.

[65] A. NETRAVALI AND B. PRASADA. Adaptive quantization of picture signals using spatial masking. *Proceedings of the IEEE*, 65(4), April 1977.

[66] ARUN N. NETRAVALI AND CHARLES B. RUBINSTEIN. Quantization of color signals. *Proceedings of the IEEE*, 65(8):1177–1187, August 1977.

[67] ARUN N. NETRAVALI AND R. SAIGAL. Optimum quantizer design using a fixed-point algorithm. *The Bell System Technical Journal*, 55(9):1423–1435, November 1976.

[68] W. M. NEWMAN AND R. F. SPROULL. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, 2nd edition, 1979.

[69] ALAN W. PAETH. Ideal tiles for shading and halftoning. In *Graphics Gems*, Paul Heckbert, editor, volume 5, chapter VIII.6, pages 486–492. Academic Press, San Diego, CA, 1995.

[70] T. N. PAPPAS AND D. L. NEUHOFF. Model-based halftoning. *Proceedings of SPIE: Human Vision, Visual Processing, and Digital Display II*, 1991.

[71] STEVE PARK. *Lecture Notes on Digital Image Processing*, chapter 7, pages 7.1–7.10. unpublished, College of William & Mary, Dept. of Computer Science, Williamsburg, VA, 1995.

[72] M. H. PIRENNE. *Vision and the Eye.* Associated Book Publishers, London, 2nd edition, 1967.

[73] CORNEL K. POKORNY AND CURTIS F. GERALD. *Computer Graphics: The principles behind the art and science.* Franklin, Beedle & Associates, Irvine, CA, 1989.

[74] DAVID POLLARD. Quantization and the method of k-means. *IEEE Transactions on Information Theory*, IT-28(2):199–205, March 1982.

[75] The persistence of vision development team, 1998.

[76] CHARLES A. POYNTON. *A Technical Introduction to Digital Video*. John Wiley & Sons, Inc, New York, 1996.

[77] WILLIAM K. PRATT. Spatial transform coding of color images. *ACM Transactions on Communications*, COM-19(6):980–992, December 1971.

[78] WILLIAM K. PRATT. *Digital Image Processing*. John Wiley & Sons, Inc, New York, NY, 1978.

[79] FRANCO P. PREPARAT AND MICHAEL IAN SHAMOS. *Computational Geometry: An Introduction*, chapter 5, pages 187–225. Springer-Verlag, New York, 1985.

[80] WERNER PURGATHOFER, ROBERT F. TOBLER, AND MANFRED GEILER. Improved threshold matrices for ordered dithering. In *Graphics Gems*, Paul Heckbert, editor, volume 5, chapter VI.1, pages 297–301. Academic Press, San Diego, CA, 1995.

[81] F. RATLIFF, H. K. HARTLINE, AND W. H. MILLER. Spatial and temporal aspects of retinal inhibitory interaction. *Journal of the Optical Society of America*, 53(1):110–120, January 1963.

[82] EVGENY ROYTMAN AND CRAIG GOTSMAN. Dynamic color quantization of video sequences. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):274–286, September 1995.

[83] D. J. SAKRISON. On the role of the observer and a distortion measure in image transmission. *ACM Transactions on Communications*, COM-25(11), November 1977.

[84] ROD SALMON AND MEL SLATER. *Computer Graphics: Systems & Concepts*. Addison-Wesley Publishing Company, New York, NY, 1987.

[85] HANAN SAMET. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[86] HANAN SAMET. *The Design and Analysis of Spatial Data Structures*, chapter 2, pages 86–105. Addison-Wesley Publishing Co., New York, NY, 1990. Describes the MX octree structure.

[87] DALE SCHUMACHER. 1-to-1 pixel transforms optimized through color-map manipulation. In *Graphics Gems*, Andrew S. Glassner, editor, volume 1, chapter 4, pages 270–274. Academic Press Professional, New York, NY, 1990.

[88] DALE A. SCHUMACHER. A comparison of digital halftoning techniques. In *Graphics Gems*, J. Arvo, editor, volume 2, chapter II.2, pages 57–71. Academic Press, San Diego, CA, 1991.

[89] M. W. SCHWARZ, W. B. COWAN, AND J. C. BEATTY. An experimental comparison of rgb, yiq, lam, hsv and opponent color models. *ACM Transactions on Graphics*, 6(2):123–158, March 1987.

[90] LOUIS D. SILVERSTEIN. Human factors for color display systems: Concepts, methods, and research. In *Color and the Computer*, H. John Durrett, editor, volume 1, chapter 2, pages 27–62. APRESS, New York, 1987.

[91] MIKE STOKES, MARK D. FAIRCHILD, AND ROY S. BERNS. Precision requirements for digital color reproduction. *ACM Transactions on Graphics*, 11(4):406–422, October 1992.

[92] MAUREEN C. STONE, WILLIAM B. COWAN, AND JOHN. C. BEATTY. Color gamut mapping and the printing of digital color images. *ACM Transactions on Graphics*, 7(4):249–292, October 1988.

[93] PETER STUCKI. Digital screening of continuous tone image data for bilevel display. In *Symposium on Advances in Digital Image Processing*, New York, 1979. Plenum Press.

[94] KALPATHI R. SUBRAMANIAN AND BRUCE F. NAYLOR. Converting discrete images to partitioning trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):273–288, July-September 1997.

[95] SPENCER W. THOMAS. Efficient inverse color map computation. In *Graphics Gems*, James Arvo, editor, volume 2, chapter III.1, pages 116–125. Academic Press, San Diego, CA, 1991.

[96] SPENCER W. THOMAS AND ROD G. BOGART. Color dithering. In *Graphics Gems*, J. Arvo, editor, volume 2, chapter II.3, pages 72–77. Academic Press, San Diego, CA, 1991.

[97] TIEN TSIN WONG AND SIU CHI HSU. Halftoning with selective precipitation and adaptive clustering. In *Graphics Gems*, Paul Heckbert, editor, volume 5, chapter VI.2, pages 302–313. Academic Press, San Diego, CA, 1995.

[98] ROBERT ULICHNEY. *Digital Halftoning*. MIT Press, Cambridge, MA, 1987.

[99] GUILLERMO VASQUEZ. Comments on "a simple approximation for minimum mean-square error symmetric uniform quantization". *IEEE Transactions on Communications*, COM-34(3):298–300, March 1986.

[100] ALEKSEJ G. VOLOBOJ. The method of dynamic palette construction in realistic visualization systems. *Eurographics*, 12(5):289–296, 1993.

[101] S. J. WAN, P. PRUSINKIEWICZ, AND S.K.M. WONG. Variance-based color image quantization for frame buffer display. *Color Research and Application*, 15(1):52–58, February 1990.

[102] S.J. WAN, S.K.M. WONG, AND P. PRUSINKIEWICZ. An algorithm for multidimensional data clustering. *ACM Transactions on Mathematical Software*, 14(2):153–162, June 1988.

[103] L. E. WEAVER. The quality rating of color television pictures. *Journal of the Society of Motion Picture Televison Engineers*, 77(6):610–612, June 1968.

[104] W. C. WILDER. Subjectively relevant error criteria for pictorial data processing. Technical Report TR-EE 72-34, Purdue University, School of Electrical Engineering, December 1972.

[105] X. WU. Efficient statistical computations for optimal color quantization. In *Graphics Gems*, James Arvo, editor, volume 2, chapter III.2, pages 126–133. Academic Press, San Diego, CA, 1991.

[106] XIAOLIN WU. Optimal bi-level quantization and its application to multilevel quantization. *IEEE Transactions on Information Theory*, 37(1):160–163, January 1991.

[107] XIAOLIN WU. Optimal quantization by matrix searching. *Journal of Algorithms*. 12:663–673, 1991.

[108] XIAOLIN WU. Color quantization by dynamic programming and principal analysis. *ACM Transactions on Graphics*, 11(4):348–372, October 1992.

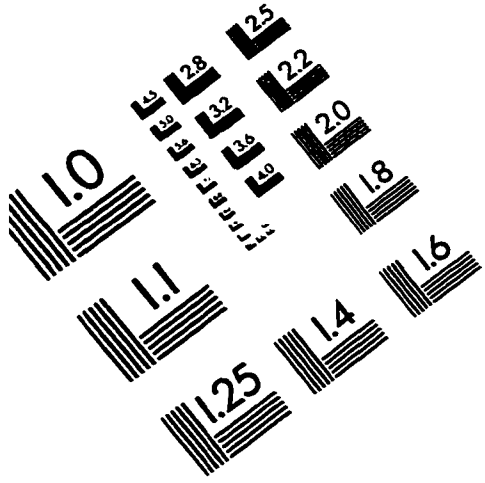
[109] G. WYSZECKI AND W. STILES. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley, New York, second edition, 1982.

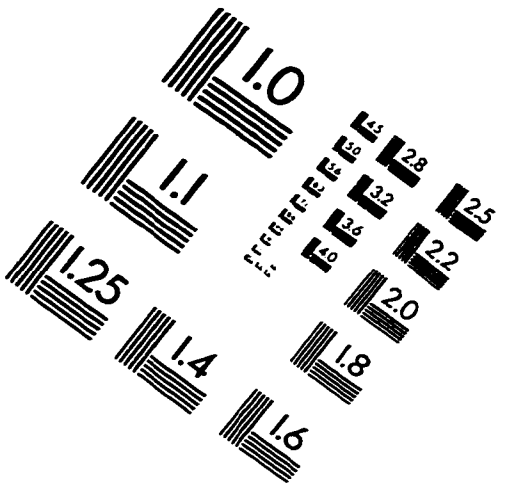
[110] ZHIGANG XIANG AND GREGORY JOY. Color image quantization by agglomerative clustering. *IEEE Computer Graphics & Applications*, 14(3):44–48, May 1994.

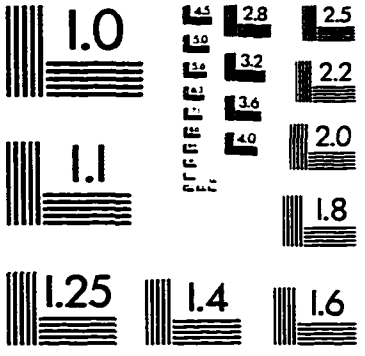
VITA

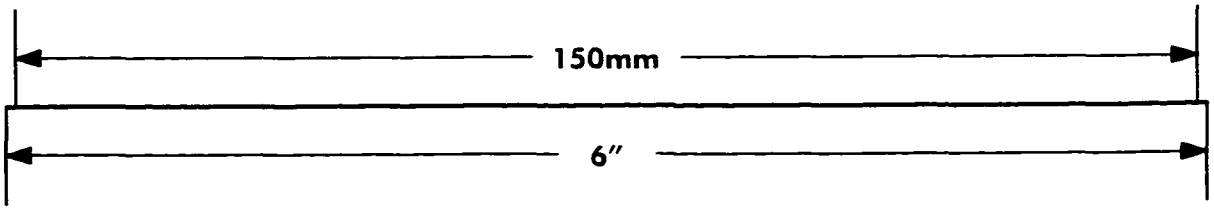
Rance Necaise was born in Gulfport, Mississippi on May 2, 1967. He graduated from Hancock North Central High School, Pass Christian, Mississippi, in 1985. Mr. Necaise attended the University of Southern Mississippi in Hattiesburg, Mississippi, where he received a B.S. degree in Computer Science in 1989 and a M.S. degree in Computer Science in 1991. He entered the College of William & Mary in August 1991 and expects to receive his doctorate in Computer Science in 1998. Mr. Necaise has held the rank of Visiting Instructor at the College of William & Mary and is currently a member of the faculty at the University of Southern Mississippi.

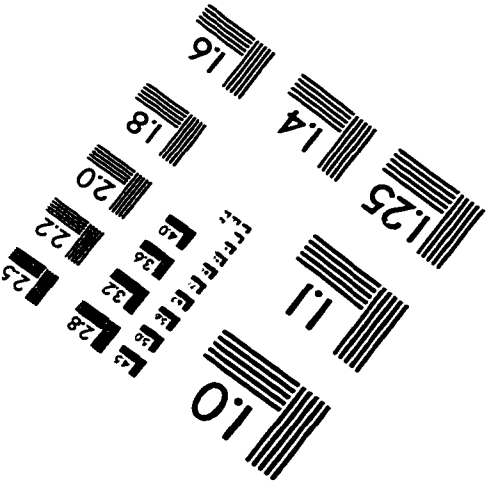
IMAGE EVALUATION
TEST TARGET (QA-3)

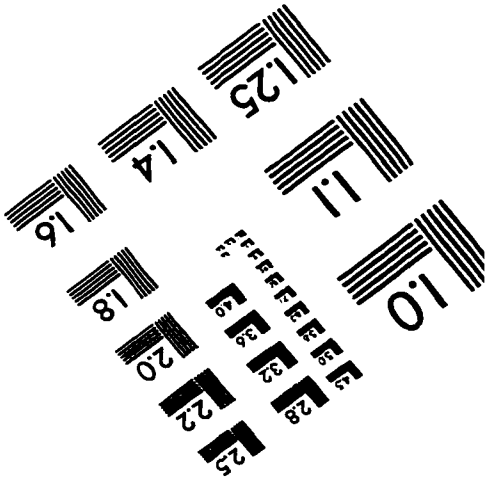












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989